Sixth Workshop on Next Generation Real-Time Embedded Systems

NG-RES 2025, January 20, 2025, Barcelona, Spain

Edited by
Patrick Meumeu Yomsi
Stefan Wildermann

Editors

Patrick Meumeu Yomsi
CISTER Research Centre, ISEP, Porto, Portugal
pmy@isep.ipp.pt

Stefan Wildermann
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Erlangen, Germany
stefan.wildermann@fau.de

ACM Classification 2012
Computer systems organization → Real-time systems; Computer systems organization → Embedded and cyber-physical systems; Software and its engineering → Software notations and tools; Networks

ISBN 978-3-95977-366-9

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-366-9.

Publication date
March, 2025

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

Digital Object Identifier: 10.4230/OASIcs.NG-RES.2025.0

ISBN 978-3-95977-366-9 **ISSN 1868-8969** **https://www.dagstuhl.de/oasics**

OASIcs – OpenAccess Series in Informatics

OASIcs is a series of high-quality conference proceedings across all fields in informatics. OASIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 1868-8969

https://www.dagstuhl.de/oasics

Contents

Papers

Sixth Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2025).
Editors: Patrick Meumeu Yomsi and Stefan Wildermann
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

On behalf of the Technical Program Committee, we are delighted to welcome you to the *Proceedings of the 6th Edition of the Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2025)*, held in Barcelona, Spain, on January 20, 2025.

The NG-RES workshop series focuses on real-time embedded systems, with special emphasis on distributed and parallel aspects. It serves as a meeting place for the networking and multicore real-time communities and fosters cross-disciplinary collaboration and innovation in embedded system design.

Key topics covered at NG-RES 2025 included, but were not limited to:

- Application of formal methods to distributed and/or parallel real-time systems
- Programming models, paradigms, and frameworks for real-time computation on parallel and heterogeneous architectures
- Dependable systems and networks
- Machine learning techniques for designing real-time systems
- Applications of approximate computing in real-time systems
- Compiler-assisted solutions for distributed and/or parallel real-time systems
- Middleware for distributed and/or parallel real-time systems
- Networking protocols and services (e.g., clock synchronization) for distributed real-time embedded systems
- Scheduling and schedulability analysis for distributed and/or parallel real-time systems
- System-level software and technologies (e.g., RTOSs, hypervisors, separation kernels, virtualization) for parallel and heterogeneous architectures

We would like to express our heartfelt gratitude to all those involved in the organization of the workshop. In particular, we would like to thank the General Chair, Federico Terraneo, and the Submission and Web Chair, Daniele Cattaneo. We would also like to express our special appreciation to the Program Committee members listed below, whose commitment and hard work were instrumental in shaping the workshop program:

- Jaume Abella Ferrer, Barcelona Supercomputing Center, Spain
- Luís Almeida, Universidade do Porto, Portugal
- Daniel Casini, Scuola Superiore Sant'Anna, Italy
- Dakshina Dasari, Robert Bosch GmbH, Germany
- Khalil Esper, Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany
- Deepak Gangadharan, International Institute of Information Technology, Hyderabad
- John Harrison Kurunathan, University of Porto, Portugal
- Ahlem Mifdaoui, University of Toulouse, France
- Marc Pouzet, École normale supérieure, Paris, France
- Mubarak Ojewale, King Abdullah University of Science and Technology (KAUST), Saudi Arabia
- Amit Kumar Singh, University of Essex, United Kingdom
- Marco Solieri, Minerva Systems and Università di Modena e Reggio Emilia, Italy
- Jürgen Teich, Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany

We also thank our publisher Schloss Dagstuhl as well as the HiPEAC organizers for contributing to the success of this workshop. Finally, we sincerely thank all the authors who contributed to NG-RES 2025. Their valuable research and insights made this workshop possible!

Patrick Meumeu Yomsi and Stefan Wildermann

Sixth Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2025).
Editors: Patrick Meumeu Yomsi and Stefan Wildermann
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

List of Authors

Pascal Becker (2)
ABB AG Corporate Research Center,
Mannheim, Germany

Nicolas Coppik (2)
ABB AG Corporate Research Center,
Mannheim, Germany

Diogo Costa (4, 5)
Centro ALGORITMI / LASI,
Universidade do Minho, Portugal

Sverre Hendseth (1)
Norwegian University of Science and Technology,
Trondheim, Norway

Erling Rennemo Jellum (1)
University of California, Berkeley, CA, USA

Edward A. Lee (1)
University of California, Berkeley, CA, USA

Shaokai Lin (1)
University of California, Berkeley, CA, USA

Marten Lohstroh (1)
University of California, Berkeley, CA, USA

José Martins (4, 5)
Centro ALGORITMI / LASI,
Universidade do Minho, Portugal

Gonçalo Moreira (4, 5)
Centro ALGORITMI / LASI,
Universidade do Minho, Portugal

Magnus Mæhlum (1)
Norwegian University of Science and Technology,
Trondheim, Norway

Afonso Oliveira (4, 5)
Centro ALGORITMI / LASI,
Universidade do Minho, Portugal

Sandro Pinto (4, 5)
Centro ALGORITMI / LASI,
Universidade do Minho, Portugal

Marcus Ritter (2)
ABB AG Corporate Research Center,
Mannheim, Germany

Martin Schoeberl (1)
Technical University of Denmark, Denmark

Jan Spieck (3)
Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU), Germany

Jürgen Teich (3)
Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU), Germany

Dominik Walter (3)
Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU), Germany

Jan Waschkeit (3)
Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU), Germany

Sixth Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2025).
Editors: Patrick Meumeu Yomsi and Stefan Wildermann
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Programming Time-Predictable Processors with Lingua Franca

Magnus Mæhlum
Norwegian University of Science and Technology, Trondheim, Norway

Erling Rennemo Jellum
University of California, Berkeley, CA, USA

Shaokai Lin
University of California, Berkeley, CA, USA

Marten Lohstroh
University of California, Berkeley, CA, USA

Martin Schoeberl
Technical University of Denmark, Denmark

Sverre Hendseth
Norwegian University of Science and Technology, Trondheim, Norway

Edward A. Lee
University of California, Berkeley, CA, USA

Abstract

Precision-timed (PRET) machines are an alternative to modern processors that provide precise control over the timing of software execution. This paper describes a platform for developing predictable real-time embedded systems that pair PRET machines with Lingua Franca (LF), a recent reactor-based coordination language with temporal semantics. Specifically, we port LF to FlexPRET, a PRET machine with flexible hardware thread scheduling. We evaluate single-threaded LF with a tight control loop style application on four embedded platforms, including the FlexPRET. The results reveal the underlying platform's timing variability and how LF plus FlexPRET can remedy this timing variability. Finally, we compare single-threaded to multithreaded LF, again concerning timing. The four embedded platforms used are FlexPRET (bare-metal), RP2040 (bare-metal), nRF52 (with Zephyr), and Raspberry Pi 3b+ (with Linux). Our results indicate that FlexPRET with LF is attractive when precise timing is essential.

2012 ACM Subject Classification Computer systems organization → Embedded and cyber-physical systems; Computing methodologies → Distributed programming languages

Keywords and phrases Real-time systems, time-predictable architecture, embedded system, coordination language

Digital Object Identifier 10.4230/OASIcs.NG-RES.2025.1

Supplementary Material

Software (Source Code): `https://github.com/magnmaeh/programming-pret-machines` [21]
archived at `swh:1:dir:8aaaf4d001e8c25c7126b4aafa70a56c9b91725c`

1 Introduction

Modern processors implement out-of-order execution, dynamic branch prediction, multi-level caching, and deep pipelines to improve average-case performance. These techniques, particularly when coupled with threaded execution and interrupts, hurt timing predictability [3, 15, 22, 25]. In real-time systems, correct program behavior is determined not only by logical function but also by its timing. Failing to reach just a single deadline can have severe consequences [10]. As such, the concern is not about average-case performance but rather worst-case execution time (WCET).

Lee et al. [13] found that engineers typically resort to some combination of the following to ensure correct execution of real-time software: (1) over-provisioning of processor capabilities, (2) using old technologies, (3) WCET analysis, and (4) real-time operating systems (RTOSes).

These techniques make it possible to develop real-time embedded systems, but they should not be necessary. The problem lies in the many layers of abstraction that discard timing semantics: instruction set architectures (ISAs), programming languages, RTOSes, and networks [3]. Precision timed (PRET) machines, and Lingua Franca (LF) are recent efforts to solve these problems.

PRET machines are a family of processors that provide cycle-accurate timers, a predictable memory hierarchy, and an interleaved pipeline [3, 13, 17]. They address the growing issue of unpredictability because timing semantics are no longer abstracted away at the ISA level. There are many members of the PRET machine family [2, 4, 7, 8, 18, 23], of which FlexPRET is a recent addition [34]. It is designed for mixed-criticality systems, providing guarantees of scheduling hard real-time tasks and flexibility for non-critical tasks. Specifically, hard real-time tasks can be temporally isolated and partially spatially isolated. Furthermore, FlexPRET's ISA contains custom timing instructions that maintain nanosecond precision.

PRET machines have been around since 2007 [3], but one major issue remains today: How are they best programmed? It is non-trivial to answer because (1) PRET machines only achieve competitive throughput when multiple hardware threads are interleaved in the pipeline [13] and (2) the programming should leverage the timing semantics available in the ISA. Previously, PRET machines have been programmed with various extensions to C [24, 31, 32]. Instead, we propose and evaluate the reactor model of computation with the polyglot coordination language Lingua Franca.

The paper is organized as follows. Section 2 provides the background on FlexPRET and LF. Section 3 presents the FlexPRET firmware extension enabling the execution of LF programs. In Section 4, we evaluate the timing characteristics of an LF program with four embedded platforms. Section 5 discusses related work. Section 6 concludes the paper.

2 Background

2.1 FlexPRET

FlexPRET is an open-source RISC-V processor with a 5-stage in-order pipeline [35]. It implements fine-grained multithreading and has an always-taken branch predictor. Scratchpad memories provide constant memory access latency as opposed to traditional cache hierarchies. It extends the RISC-V ISA with custom timing instructions to enable the developing of time-sensitive applications.

FlexPRET fetches instructions from different threads during each cycle. The instructions are interleaved in the pipeline with instructions from other *hardware threads*. Each hardware thread has its program counter, register file, and other control registers. A hardware scheduler manages which thread to run at each clock cycle. With multiple threads interleaved in the pipeline, the spacing between dependent instructions increases, which reduces the number of pipeline bubbles. This increases the overall throughput.

FlexPRET's hardware threads can be configured as a hard real-time thread or a soft real-time thread. Hard real-time threads have temporal isolation, meaning they can guarantee a fraction of all processor cycles independent of the execution of other threads. When a hard real-time thread finishes its task, it may enter a sleeping state, which frees up its cycles to be used by soft real-time threads. In a multicore system, the user might assign execution of the main application to core 0 and interrupt handling to core 1 to provide temporal isolation for core 0. FlexPRET can achieve the same by assigning the main application to hardware thread 0 and interrupt handling to hardware thread 1.

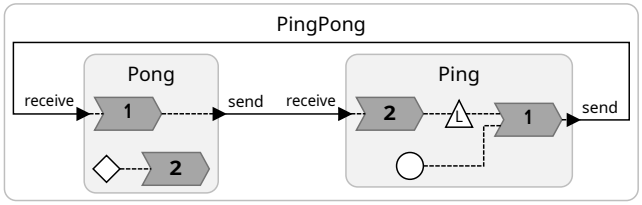


Figure 1 An auto-generated diagram representing an example program.

2.2 Lingua Franca

Lingua Franca (LF) is a polyglot coordination language introducing determinism to concurrent and distributed systems [19] by using actors exchanging messages with time stamps [20]. Figure 1 shows an auto-generated diagram representing a LF program. In LF, a program comprises stateful components called *reactors* represented as rounded rectangles in the diagram. In Figure 1 there are three reactors: the `Ping` reactor, the `Pong` reactor, and the top level `main` reactor. In this LF program, the `Ping` reactor sends integers to the `Pong` reactor, which echos them back.

Reactors encapsulate event-triggered *reactions* represented as dark grey chevrons in the diagram. The reactions encapsulate a sequential program called the reaction body, written in the target language. A reaction is associated with a set of *triggers* and *effects*. Triggers include input ports of the parent reactor, represented as black triangles; actions, represented as white triangles; timers, represented as clocks; and the two builtin triggers *startup* and *shutdown*, represented by a white circle and a white diamond, respectively. Triggers can be typed, meaning that their events also carry a payload. E.g., the input port `receive` of `Ping` has the type `int`. In the diagram, a dashed line between a reaction and a trigger or an effect indicates a dependency.

A reaction is invoked whenever there is an event at any of its triggers, and it might produce events that affect any of its effects, which include output ports and actions. In LF, it is the responsibility of the runtime to execute the reactions such that all of their dependencies are met, and determinism is assured. This relieves the programmer from handling low-level synchronization primitives such as locks, semaphores, and condition variables. Reactors communicate through `connections`, drawn as solid lines connecting input and output ports. An input port can only be driven by a single upstream output port.

LF has a semantic notion of logical time. Each event has a tag that denotes the logical time it should be handled. The logical time of an event imposes a lower bound on the physical time at which it is handled. A deadline can be associated with a reaction to impose an upper bound. It specifies the maximum allowed discrepancy between the logical and the physical time at which the reaction is invoked. A violation will trigger the deadline miss handler supplied by the programmer.

LF introduces determinism to concurrent and distributed software, a property rarely found in these systems [10, 12, 14]. Given an initial state and a given input data to an LF program, LF imposes a partial order on the triggering reactions to ensure data determinism. Computed data will not depend on scheduling decisions nor on execution times (unless deadlines are violated). We refer to this as *logical* determinism. PRET machines have *temporal* determinism. Given its input data, a program executed on a PRET machine defines a unique timing behavior. When PRET machines are programmed with LF, we can expand the model of computation to encompass temporal *and* logical determinism. As such, the correctness of the system is determined not only by the ordering of outputs but also by their timing.

```
target C;
reactor Sensor {
  output out: int
  physical action a(0 msec, 10 msec):int
  reaction(a) -> out {=
    int filtered = filter(a->value);
    lf_set(out, filtered);
  =}
}
reactor Actuator {
  input in: int
  reaction(in) {=
    // Handle filtered sensor message.
  =} deadline(5 msec) {=
    // Handle deadline violation.
  =}
}
main reactor {
  sensor = new Sensor()
  act = new Actuator()
  sensor.out -> act.in
}
```

Listing 1 A LF program reaction to asynchronous events with a deadline.

LF is particularly suited for the programming of PRET machines because:

(1) It has a deterministic semantics establishing well-defined ordering constraints on processing events. As such, we move away from programming PRET machines with threads.

(2) It simplifies concurrent development and exploits its knowledge of independent reactors to execute them in parallel. This increases the overall throughput of a PRET machine.

(3) It has a semantic notion of time, which lets the developer specify time-sensitive tasks, enforce deadlines, and handle deadline misses. This can directly map to semantics in a PRET machine's ISA, which yields efficient implementation.

(4) It is based on a scalable programming model that can efficiently be executed on multicore and distributed systems. For future work, we intend to extend LF support to InterPRET [8], a multicore PRET machine based on FlexPRET.

3 Lingua Franca for FlexPRET

In this section, we present FlexPRET's LF runtime support. We begin by motivating the use of reactor-oriented programming for PRET machines before delving into the details of supporting single-threaded and multithreaded LF programs.

3.1 Motivation

Our overarching goal is to develop predictable real-time systems. PRET machines are time-predictable on the microarchitectural level and, as such, lay the foundation for our goal. Most real-time systems will have concurrent tasks, each performing different functions with different timing requirements. This is also matched by the PRET machines' microarchitectural support for concurrency through hardware threads. However, if there is any interdependency

```
reaction(a) -> out {=
    int filtered = a->value;
    fp_interrupt_on_expire(lf_time_logical() + MSEC(5), timeout);
    filtered = filter(a->value);
    fp_interrupt_on_expire_cancel();
timeout:
    lf_set(out, filtered);
=}
```

Listing 2 An LF program that reacts to asynchronous events with a deadline.

among the different concurrent tasks executing on the various threads, ensuring deterministic execution becomes challenging. Typically, programmers rely on mutexes, semaphores, and condition variables to orchestrate such tasks, all of which are error-prone techniques.

LF addresses these issues and provides deterministic coordination of concurrent real-time tasks. A network of reactors will denote, at compile time, a well-defined ordering of reaction executions. However, to deliver real-time computations in LF, we must be able to provide an upper bound on the execution times of reactions.

Consider the LF program shown in Listing 1. It consists of two reactors. `Sensor` has a physical action `a` that is triggered by an asynchronous event with a `min_delay` of 0 and a `min_spacing` of 10 ms. The events are filtered and passed to `Actuator` that reacts with a 5 ms deadline. If this is a hard real-time system, we must guarantee that the `Actuator` does not miss any deadlines. By running this program on a FlexPRET, it is possible to derive a tight bound on the WCET of the reaction in `Sensor`. Suppose we can compute a WCET of less than 5 ms (neglecting the runtime overhead). In that case, we can statically guarantee the latency from handling the asynchronous input until triggering the `Actuator` reaction. However, this alone is insufficient; we might miss deadlines if the asynchronous events arrive too frequently. In LF, this is handled by the `min_spacing` argument on the physical action, which creates a lower bound on the time between events. The LF runtime handles events that violate this bound according to some policy (the events are either deferred, dropped, or replaced). In the case of Listing 1, we are guaranteed not to receive more than one asynchronous event per 10 ms. Not all reaction bodies will allow for deriving a sufficiently small WCET; for some reaction bodies, no WCET can be derived. With FlexPRET, we can easily handle this using the `interrupt_on_expire` instruction combined with `setjmp` and `longjmp`. The reaction of `Sensor` can be rewritten as shown in Listing 2. Now, the reaction is guaranteed to terminate within 5 ms even if the execution time of the function `filter` is greater than 5 ms. This programming style can be dangerous because it can leave a program in an inconsistent state, and any updates to shared global variables should be done with interrupts disabled.

3.2 Single-Threaded Lingua Franca Support

Figure 2 shows an overview of the software stack executing LF programs on the FlexPRET platform. At the top of the stack is the *LF application* written by the user. It builds on both the *LF runtime* and the *FlexPRET library*, as the user may opt-in to use functionality from the FlexPRET libraries directly. The LF runtime exposes an application programming interface (API) to the LF application with functions for reading and writing from ports, reading the current logical and physical time, and scheduling new events.

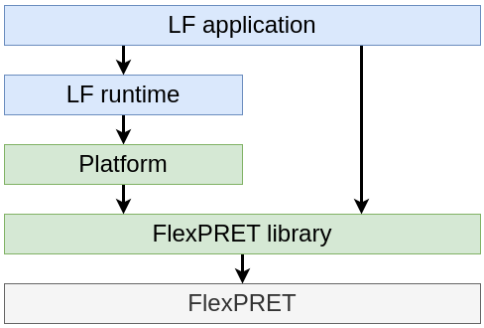


Figure 2 LF's software stack for FlexPRET.

The LF runtime itself is built in terms of a generalized API implemented in *Platform*, with functions and type definitions. Examples are `void lf_initialize_clock(void)` or `int lf_critical_section_enter(void)`. Adding support for a platform means implementing these functions for the particular platform. Since FlexPRET has native support for timing functionality, many implementations directly map to its custom instructions, which yields efficient implementations.

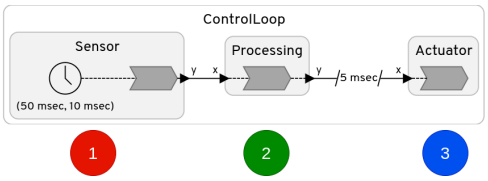
The LF C runtime can run in single-threaded or multithreaded modes. In the single-threaded mode, the LF runtime executes on a single FlexPRET hardware thread. The other hardware threads can be used for different tasks, such as interrupt handling. The two are temporally isolated by running the LF application on one hardware thread and interrupt handling on another. This protects the timing characteristics of the LF application. Assuming they do not share memory, they are isolated, as if running on two isolated processors. Another possibility is to run an entirely different program, possibly written in LF.

3.3 Multithreaded Lingua Franca Support

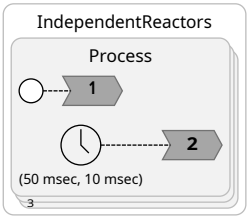
In the multithreaded mode, the LF runtime coordinates multiple workers, each executing within a thread. FlexPRET exposes *hardware* threads to the LF scheduler instead of the usual *software* threads. Context switching between hardware threads is round-robin every clock cycle, unlike software threads that typically have a millisecond granularity. This gives the single-core FlexPRET the appearance of executing several threads *in parallel*, like a multicore processor. Multiple FlexPRET hardware threads can control time-sensitive applications involving multiple actuators simultaneously.

However, FlexPRET cannot support more than N worker threads, where N is the number of hardware threads available on the core (N is 8 in current implementations). Software threads and a scheduler can be built on top of one or more hardware threads to increase the number of threads, but this defeats the purpose of the PRET machine. Fortunately, LF delivers data determinism for any number of worker of threads.

Supporting multithreaded LF on FlexPRET requires additional function implementations, primarily related to running and synchronizing threads. FlexPRET implements all thread synchronization through a single hardware lock (a single bit wide). Acquiring and releasing the lock are atomic operations, and only a single thread may hold the lock at any time. Because of this, the lock can compose any number of operations into an atomic one. We use the single hardware lock to implement other thread synchronization primitives, such as mutexes, semaphores, and conditional variables. Because the critical sections using the hardware lock are very short, we implement it as a spinlock.



(a) A tight control loop benchmark. The colors are legends for the upcoming histograms.



(b) Independent processing benchmark. There are no data dependencies among the reactors.

Figure 3 The LF benchmark programs we use in the evaluation.

We found that FlexPRET's hardware lock mechanism can become a bottleneck. The multithreaded LF runtime relies on many synchronization primitives, which may cause a lot of congestion. Multithreaded LF will be more suitable once FlexPRET adds better hardware support for thread synchronization, e.g., like in Strøm et al. [30]

4 Evaluation

The evaluation consists of two parts. In Section 4.1-4.2, we implement a tight control loop in LF in single-threaded mode and benchmark its timing characteristics. We compare the results across four different embedded platforms. A complete description of the setup can be found on Github.[1] In Section 4.3, we compare the single-threaded and multithreaded LF runtime for FlexPRET with other commonly used embedded platforms.

4.1 Benchmark Description

Control systems often run with a fixed period and consist of (1) the sampling of sensors, (2) processing and sensor fusion, and (3) driving of actuators. Variations in the period between two invocations of the same stage are known as jitter. Jitter can harm control system performance concerning actuation (i.e., output jitter) and sampling (i.e., input jitter).

In the first experiment, we evaluate the timing characteristics of a tight control loop shown in Figure 3a. We sample the time when the Sensor reactor transmits a measurement to the Processing stage (1). Next, we sample the time to run the Processing stage (2). Like (1), we sample when the Actuator receives a command from the Processing stage (3).

The stages in the control loop are simulated. The Sensor samples a normal distribution and transmits it to the Processing stage. In general, the execution time of control algorithms may depend on the sensor data, e.g., if the control algorithm runs an optimization. To simulate this behavior, we implement the Processing stage as `for (volatile int i = 0; i < x; i++);` where x is the measurement from the sensor. This introduces jitter to the Processing stage. We then leverage LF's *delayed connections* to decouple the connection between Processing and Actuator, which adds a logical delay to the data to remove the jitter from the earlier stages, as in the logical execution time (LET) principle [5, 6].

We evaluate the control loop on four different embedded platforms. Table 1 gives some key parameters for each platform. FlexPRET is synthesized for a Zedboard Zynq-7000 Field Programmable Gate-Array. The RP2040 has a default clock frequency of 133 MHz, but is

[1] `https://github.com/magnmaeh/programming-pret-machines`

Table 1 An overview of all four platforms evaluated.

Platform	#cores@freq	Abstraction layer	Processor(s)
FlexPRET	1@100MHz	Bare-metal	RISC-V custom
RP2040	1@64MHz	Bare-metal	ARM Cortex-M0+
nRF52dk_nRF52832	1@64MHz	Zephyr RTOS [9]	ARM Cortex-M4F
Raspberry Pi 3b+	4@1.4GHz	Raspberry Pi OS (formerly Raspbian)	ARM Cortex-A53

reduced to 64 MHz to make it comparable to nRF52. The Raspberry Pi 3b+ (RPi) operating system has a default configuration and runs some services in the background. We have made no effort to optimize RPi for real-time applications; this is outside the scope of this paper.

To evaluate the robustness of each platform, we trigger interrupts while the benchmark runs. We distinguish between *periodic* and *sporadic* interrupts. We transmit the periodic interrupts with a fixed period of 10.5 ms, which is selected not to match the period of the tight control loop, which is 10 ms. The periodic interrupts could emulate, e.g., a sensor interrupt. In addition, we transmit interrupts sporadically and in bursts, which simulate a network connection. The periodic and sporadic interrupts have an interrupt handler, which performs a fixed amount of computation. Depending on which platform is used, this might disrupt the timing characteristics of the benchmark. All platforms receive identical interrupt signals, and throughout the benchmark, we transmit approximately 1000 periodic interrupts and 100 sporadic interrupts.

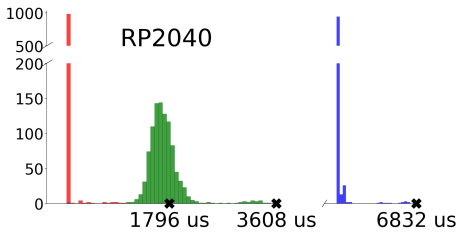
The physical setup is shown in the code repository.[2] After completing the benchmark, a computer uploads executables to the target platform and receives timestamps. We use a Digilent Analog Discovery as a waveform generator. It transmits periodic and sporadic interrupt signals after receiving a trigger signal from the platform. The computer predetermined the exact waveforms, making it possible to transmit identical waveforms to all platforms.

4.2 Benchmark Results

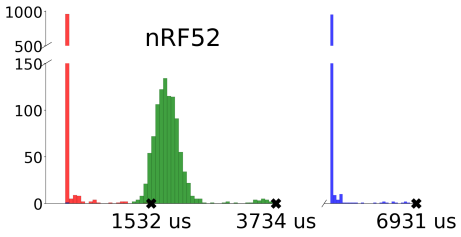
Figure 4 shows the results for all platforms. The RP2040 and nRF52 are comparable platforms and yield similar results. They both have long tails behind the most recurring execution times from interrupt handling. In Figures 4a and 4b, we mark three points on the x-axis, each corresponding to the worst observed execution time of each stage in the control loop. E.g. for the RP2040, the worst observed execution time for the Sensor stage is 1796 us and 3608 us for the Processing stage. However, when no interrupts occur, the Sensor and Actuator stages execute within 100 us. Therefore, interrupts significantly impact the execution time of the reactions. As expected, we conclude that the RP2040 and nRF52 platforms are susceptible to jitter from interrupt handling.

As seen in Figure 4c, the RPi does not have long tails like the RP2040 and nRF52 but instead has a few rare outlier points. The RPi has four processor cores and runs interrupt handlers on a core different from the one running the application. Therefore, the timing characteristics of the benchmark are unaffected by interrupts. The outlier points instead come from the operating system (OS), such as running background services or handling page faults. The programmer cannot control such OS issues without changing the underlying OS, and it is difficult to determine how they impact the timing characteristics of the application. In Figure 4c, the worst observed execution time for the Sensor stage is 3037 us, almost 3000 us

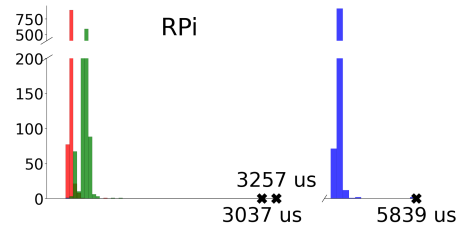
[2] `https://github.com/magnmaeh/programming-pret-machines`



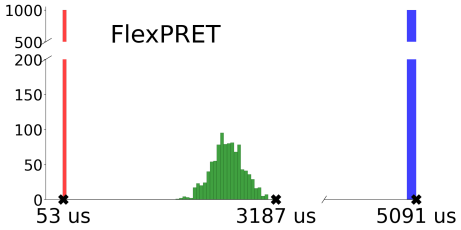
(a) RP2040 has a long "tail" behind the most recurring execution times stemming from interrupt handling.



(b) nRF52 has a long "tail" behind the most recurring execution times stemming from interrupt handling.



(c) RPi has some jitter in all stages. It is unaffected by interrupts, but operating system features introduce jitter.



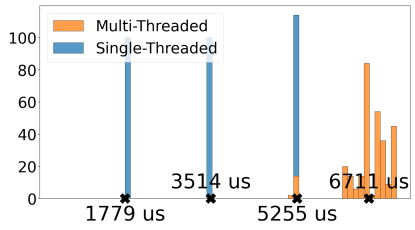
(d) FlexPRET has no jitter in the Sensor and Actuator stages. Interrupts introduce no jitter. However, it spends a relatively long time inside the Processing stage.

Figure 4 The results from running the tight control loop benchmark in Figure 3a on the different platforms. The x-axis is broken from the maximum time measured in the Processing stage until approximately 5000 μs. The width of the bars has no meaning; they can appear wider because we 'zoom' into certain intervals on the x-axis.

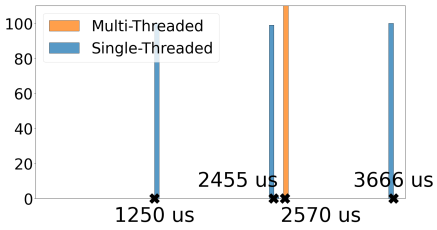
more than the average case. However, the outlier in the Actuator stage is much less severe, only adding approximately 800 us to the average case. It is difficult to determine whether we will observe even worse execution times. We conclude that the RPi is mainly immune from the jitter introduced by interrupt handling, but the underlying platform (including OS) is inherently unpredictable.

Figure 4d shows the results for FlexPRET. FlexPRET has temporal isolation between the benchmark application and the interrupt handlers, much like the RPi running the interrupt handlers on idle processor cores. Unlike the RPi, however, the FlexPRET is programmed in bare-metal and does not suffer from OS timing issues. On the other hand, the FlexPRET spends a relatively long time inside the Processing stage. This is a combination of a relatively low clock frequency of 100 MHz and using multiple hardware threads. The hardware thread running the benchmark is configured as a hard real-time thread with a schedule granting it 1/4 of all processor cycles, resulting in effectively 25 MHz. Aside from the relatively slow execution, FlexPRET has no jitter in the Sensor and Actuator stages. The jitter in the Processing stage is inserted by design, and it does not propagate to the Actuator stage due to a delayed connection from LF.

Two takeaways from Figures 4a-4d: First, LF applications on FlexPRET run with negligible jitter compared to the other platforms. On the FlexPRET, the execution time of the Sensor stage is exactly 47.280 us *every* iteration of the control loop, except for the first iteration due to initialization of the LF runtime. In contrast, the RP2040 and nRF52 have moderate jitter in these stages when no interrupts trigger and extreme jitter when they do. Second, despite the jitter stemming from the underlying platform or data-dependent processing, the LF's delayed connection successfully filters out the jitter from the Sensor and Processing stages.



(a) The results for nRF52.



(b) The results for FlexPRET. The height of the multithreaded bar is 300.

Figure 5 The results from running the independent reactors benchmark from Figure 3a in unthreaded and threaded mode for FlexPRET and nRF52.

4.3 Multithreaded Lingua Franca

To compare LF in single-threaded and multithreaded mode running on the FlexPRET, we implement a simple LF application consisting of three identical reactors as shown in Figure 3b. Each reactor contains a single reaction that performs a long, constant-time computation. In LF, the three reactions will execute logically simultaneously. However, the actual physical execution timeline will depend on the platform. We execute this program in multithreaded and single-threaded modes on the FlexPRET and nRF52 with Zephyr.

Figure 5a shows the results for nRF52. The x-axis displays the time. With the single-threaded LF runtime, the three reactions execute sequentially. The reactions finish at different physical times: 1779 us, 3514 us, and 5255 us. In addition, the LF runtime may execute the reactions in any order. This means a given reactor might finish at very different physical times when executed twice, e.g., at 1779 us one iteration and at 5255 us another iteration.

Using the multithreaded LF runtime on the nRF52 adds much overhead. We configure Zephyr's scheduler to be preemptive with a time slice of 1 ms. This means any running thread must yield execution to other threads after 1 ms. We also configure LF to use three worker threads. From Figure 5a we can see that using the multithreaded runtime adds quite a lot of overhead; this is a combination of overhead from LF multithreaded runtime, overhead from Zephyr's scheduling algorithm running every 1 ms, and the coarse granularity of the system clock that triggers the scheduler. However, there is less variation in the physical times the reactions finish.

Next, we consider the results for FlexPRET in Figure 5b. Similar to the nRF52, the single-threaded LF runtime executes the reactions sequentially. The FlexPRET finishes execution quicker than the nRF52 because its clock frequency is 100 MHz instead of 64 MHz, and the FlexPRET's hardware thread executing the LF application is given all clock cycles. (In the previous benchmark, the LF application was given only 1/4 of all clock cycles, essentially running it at 25 MHz.) Similar to the nRF52, sequential execution of the reactions means they finish at widely different physical times.

Changing to the multithreaded LF runtime has two consequences. First, all reactions finish execution at approximately the same time. They appear to finish simultaneously in the plot, but in reality, the execution time is distributed around 2570 us with a standard deviation of 7.8 us. The small amount of jitter probably comes from FlexPRET's hardware synchronization primitives. Second, the total time to execute all three reactions decreases. As established in Section 2.1, interleaving multiple hardware threads in the pipeline increases FlexPRET's overall throughput. The effect is strengthened by the aforementioned long computation being implemented as `for (volatile int i = 0; i < 10000; i++);`. This compiles into frequent load and branch instructions, the latter being the culprit of pipeline bubbles in single-threaded execution.

5 Related Work

MultiPRET [7] is a multicore version of FlexPRET using shared memory for inter-core communication. The authors provide runtime support for the synchronous language ForeC [33].

XCore [23] is a commercial multicore PRET architecture developed by XMOS. It is also based on a barrel register file with fine-grained multithreading of hardware threads. Unlike MultiPRET, XCore does inter-core communication using a credit-based network-on-chip. The XCores can be programmed with a C-like language called xC, which has semantics based on Communicating Sequential Processes (CSP) [31].

InterPRET [8] is another multicore PRET machine based on FlexPRET. Like the XCore, it is based on a network-on-chip for inter-core communication. Unlike XCore, however, InterPRET is based on a time-triggered, time-predictable network-on-chip [28]. This opens the door for end-to-end time-predictable computations on a many-core processor. InterPRET is currently programmed in C, but we intend to leverage the FlexPRET support for LF to program multicore PRET machines in LF.

Patmos is, like the PRET architecture, a processor designed for real-time systems [29]. For example, the caches are optimized for time-predictability. Like InterPRET, several Patmos processors are combined with a network-on-chip to build the T-CREST multicore architecture [26]. Patmos supports the single-threaded LF runtime [11], including WCET analysis of LF reactions [27]. We are currently developing the multithreaded LF runtime for the multicore T-CREST architecture.

The logical execution time (LET) model, which can be traced back to the time-triggered language Giotto [6], is a related approach that is gaining popularity in the automotive sector and was recently included in the AUTOSAR standard [1]. System-level LET [5] is a recent extension to LET, introducing Time Zones and interconnecting LET, enabling the modeling of distributed systems using LET. The time-predictability of PRET machines makes them an ideal fit for LET-style programming. We focus on LF since it can be considered a generalization of the LET model [16] and provides compilation and runtime infrastructure for both modeling and implementing LET systems.

6 Conclusion

This paper presents a new platform for real-time systems based on the reactor-oriented programming of PRET machines. We present runtime support for executing Lingua Franca programs on FlexPRET. We evaluate LF programs on FlexPRET and several other commonly used embedded platforms. Regarding timing, LF on FlexPRET outperforms the other platforms in single-threaded and multithreaded execution. We believe this opens up an interesting design space and can be the basis for promising future work.

References

1 AUTOSAR. Specification of timing extensions. *AUTOSAR CP Release 4.4.0*, October 2018.

2 M. Delvai, W. Huber, P. Puschner, and A. Steininger. Processor support for temporal predictability - the SPEAR design example. In *15th Euromicro Conference on Real-Time Systems, 2003. Proceedings.*, pages 169–176, 2003. doi:10.1109/EMRTS.2003.1212740.

3 Stephen Edwards and Edward A. Lee. The case for the precision timed (pret) machine. Technical Report UCB/EECS-2006-149, EECS Department, University of California, Berkeley, November 2006. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-149.html.

4 Stephen A. Edwards, Sungjun Kim, Edward A. Lee, Isaac Liu, Hiren D. Patel, and Martin Schoeberl. A disruptive computer design idea: Architectures with repeatable timing. In *Proceedings of IEEE International Conference on Computer Design (ICCD 2009)*, Lake Tahoe, CA, October 2009. IEEE.

5 Kai-Björn Gemlau, Leonie Köhler, Rolf Ernst, and Sophie Quinton. System-level logical execution time: Augmenting the logical execution time paradigm for distributed real-time automotive software. *ACM Trans. Cyber-Phys. Syst.*, 5(2), January 2021. doi:10.1145/3381847.

6 T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. In *EMSOFT 2001*, volume LNCS 2211, pages 166–184. Springer-Verlag, 2001.

7 Nicolas Hili, Alain Girault, and Eric Jenn. Worst-case reaction time optimization on deterministic multi-core architectures with synchronous languages. In *2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–11, 2019. doi:10.1109/RTCSA.2019.8864570.

8 Erling R. Jellum, Shaokai Lin, Peter Donovan, Chadlia Jerad, Edward Wang, Marten Lohstroh, Edward A. Lee, and Martin Schoeberl. Interpret: A time-predictable multicore processor. In *Proceedings of Cyber-Physical Systems and Internet of Things Week 2023*, CPS-IoT Week '23, pages 331–336, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3576914.3587497.

9 Erling Rennemo Jellum, Shaokai Lin, Peter Donovan, Efsane Soyer, Fuzail Shakir, Torleiv Bryne, Milica Orlandic, Marten Lohstroh, and Edward A. Lee. Beyond the Threaded Programming Model on Real-Time Operating Systems. In Federico Terraneo and Daniele Cattaneo, editors, *Fourth Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2023)*, volume 108 of *Open Access Series in Informatics (OASIcs)*, pages 3:1–3:13, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.NG-RES.2023.3.

10 Mike Jones. What really happened on Mars Rover Pathfinder. Technical report, Microsoft, 1997.

11 Ehsan Khodadad, Luca Pezzarossa, and Martin Schoeberl. Towards lingua franca on the patmos processor. In *Proceedings of the 2024 IEEE 27th International Symposium on Real-Time Distributed Computing (ISORC)*, 2024.

12 Phil Koopman. A case study of toyota unintended acceleration and software safety. Technical report, ECE Department, Carnegie Mellon University, 2014.

13 Edward Lee, Jan Reineke, and Michael Zimmer. Abstract PRET machines. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 1–11, 2017. doi:10.1109/RTSS.2017.00041.

14 Edward A. Lee. The problem with threads. *IEEEC*, 39(5):33–42, 2006. doi:10.1109/MC.2006.180.

15 Edward A. Lee. Computing needs time. *Communications of the ACM*, 2009.

16 Edward A. Lee and Marten Lohstroh. *Generalizing Logical Execution Time*, pages 160–181. Springer Nature Switzerland, Cham, 2022. doi:10.1007/978-3-031-22337-2_8.

17 Isaac Liu. *Precision Timed Machines*. PhD thesis, EECS Department, University of California, Berkeley, May 2012. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-113.html.

18 Isaac Liu, Jan Reineke, David Broman, Michael Zimmer, and Edward A. Lee. A PRET microarchitecture implementation with repeatable timing and competitive performance. In *Proceedings of the 30th IEEE International Conference on Computer Design (ICCD 2012)*, October 2012. URL: http://chess.eecs.berkeley.edu/pubs/919.html.

19 Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. Toward a lingua franca for deterministic concurrent systems. *ACM Transactions on Embedded Computing Systems*, 2021.

20 Marten Lohstroh, Martin Schoeberl, Andrés Goens, Armin Wasicek, Christopher Gill, Marjan Sirjani, and Edward A. Lee. Actors revisited for time-critical systems. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, pages 152:1–152:4, New York, NY, USA, June 2019. ACM. doi:10.1145/3316781.3323469.

21 Magnus Mæhlum. Programming PRET machines. Software, swhId: swh:1:dir:8aaaf4d001e8c25c7126b4aafa70a56c9b91725c (visited on 2025-03-17). URL: https://github.com/magnmaeh/programming-pret-machines, doi:10.4230/artifacts.23000.

22 Goncalo Martins, Dave Lacey, Allistair Moses, Matthew J. Rutherford, and Kimon P. Valavanis. A case for i/o response benchmarking of microprocessors. In *IECON 2012 - 38th Annual Conference on IEEE Industrial Electronics Society*, pages 3018–3023, 2012. doi:10.1109/IECON.2012.6389416.

23 David May. *The XMOS XS1 Architecture*, 2009.

24 Saranya Natarajan and David Broman. Timed c: An extension to the c programming language for real-time systems. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 227–239, 2018. doi:10.1109/RTAS.2018.00031.

25 Martin Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, vol. 2009, Article ID 758480:17 pages, 2009. doi:10.1155/2009/758480.

26 Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva, Jens Sparsø, and Alessandro Tocchi. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015. doi:10.1016/j.sysarc.2015.04.002.

27 Martin Schoeberl, Ehsan Khodadad, Shaokai Lin, Emad Jacob Maroun, Luca Pezzarossa, and Edward A. Lee. Invited Paper: Worst-Case Execution Time Analysis of Lingua Franca Applications. In Thomas Carle, editor, *22nd International Workshop on Worst-Case Execution Time Analysis (WCET 2024)*, volume 121 of *Open Access Series in Informatics (OASIcs)*, pages 4:1–4:13, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.WCET.2024.4.

28 Martin Schoeberl, Luca Pezzarossa, and Jens Sparsø. S4noc: a minimalistic network-on-chip for real-time multicores. In *Proceedings of the 12th International Workshop on Network on Chip Architectures*, pages 8:1–8:6. ACM, October 2019. doi:10.1145/3356045.3360714.

29 Martin Schoeberl, Wolfgang Puffitsch, Stefan Hepp, Benedikt Huber, and Daniel Prokesch. Patmos: A time-predictable microprocessor. *Real-Time Systems*, 54(2):389–423, April 2018. doi:10.1007/s11241-018-9300-4.

30 Tórur Biskopstø Strøm, Jens Sparsø, and Martin Schoeberl. Hardlock: Real-time multicore locking. *Journal of Systems Architecture*, 97:467–476, 2019. doi:10.1016/j.sysarc.2019.02.003.

31 Douglas Watt. *Programming XC on XMOS devices*, 2009.

32 Eugene Yip, Alain Girault, Partha S. Roop, and Morteza Biglari-Abhari. The forec synchronous deterministic parallel programming language for multicores. In *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*, pages 297–304, 2016. doi:10.1109/MCSoC.2016.13.

33 Eugene Yip, Alain Girault, Partha S. Roop, and Morteza Biglari-Abhari. Synchronous deterministic parallel programming for multi-cores with ForeC. *ACM Transactions on Programming Languages and Systems*, 2023.

34 Michael Zimmer. *Predictable Processors for Mixed-Criticality Systems and Precision-Timed I/O*. PhD thesis, EECS Department, University of California, Berkeley, August 2015. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-181.html.

35 Michael Zimmer, David Broman, Chris Shaver, and Edward A. Lee. FlexPRET: A processor platform for mixed-criticality systems. *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014. doi:10.1109/RTAS.2014.6925994.

Low-Latency Real-Time Applications on Heterogeneous MPSoCs

Nicolas Coppik
ABB AG Corporate Research Center, Mannheim, Germany

Pascal Becker
ABB AG Corporate Research Center, Mannheim, Germany

Marcus Ritter
ABB AG Corporate Research Center, Mannheim, Germany

Abstract

Heterogeneous Multi-Processor Systems-on-Chip (MPSoCs) that combine multiple, heterogeneous processing units are becoming increasingly popular for a wide range of applications, including industrial applications, where complex real-time applications can benefit from the performance and flexibility they offer.

However, deploying real-time applications with low latency requirements across multiple processing units on such MPSoCs remains a challenging problem, particularly when communication between processors is required on a time-critical path. Existing solutions generally rely on the presence of at least one full-featured, general-purpose operating system on the device, and do not cater to the requirements of distributed, low-latency real-time applications.

In this paper, we investigate the performance, with a focus on latency, of different options for communication between CPUs, including inter-processor interrupts and shared memory communication via different memories, on the popular Xilinx Zynq UltraScale+ platform and propose a novel solution for communication between heterogeneous processing units that relies only on the availability of shared memory. Our solution is capable of achieving sub-microsecond latencies for signaling and the transfer of small amounts of data between processing units, making it suitable for deploying distributed, low-latency real-time applications.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computer systems organization → Embedded software; Computer systems organization → Multicore architectures

Keywords and phrases real-time systems, heterogeneous systems, latency, inter-core communication

Digital Object Identifier 10.4230/OASIcs.NG-RES.2025.2

1 Introduction

Embedded systems used in industrial applications are becoming both more performant and increasingly complex. As part of this trend, heterogeneous Multi-Processor Systems-on-Chip (MPSoCs) are gaining popularity. These devices typically combine multiple, heterogeneous CPUs with other computational resources, such as GPUs or FPGAs. This makes them well-suited to a wide range of applications, and popular in domains like industrial or automotive applications. A common example is the Xilinx Zynq UltraScale+ family of MPSoCs, which include both a multi-core ARM Cortex-A53 CPU as well as a dual-core ARM Cortex-R5 CPU, the latter of which is suitable for running real-time applications which require deterministic performance, while the former can be used for more computationally intensive applications.

However, in a parallel trend, real-time applications, for instance in industrial applications, have also grown more complex and computationally expensive, to the point where the individual CPUs on an MPSoC may not be sufficient for meeting their performance requirements, and distributing them across the different, heterogeneous CPUs becomes necessary.

While frameworks and libraries like OpenAMP[1] to facilitate the development applications for heterogeneous MPSoCs exist, they are most commonly used in configurations that include a full-featured, general-purpose OS on one of the CPUs and a real-time operating system on another (for example, running Linux and FreeRTOS on the Cortex-A53 and Cortex-R5 cores of a Xilinx Zynq UltraScale+ device). They are not suitable for low-latency real-time applications.

In this paper, we investigate the performance of different hardware primitives for communicating between different processing units on the Xilinx Zynq UltraScale+ MPSoC, identify an approach that relies exclusively on shared memory as the most suitable, and develop a solution for inter-processor communication in low-latency real-time applications distributed across heterogeneous processing units on top of it.

This paper is structured as follows: We first provide an overview of the system we are working with and the requirements our solution is intended to meet in Section 2. Benchmarking results are presented in Section 3. We describe our solution in Section 4, summarize related work in Section 5. and provide concluding remarks in Section 6.

2 System Overview

We consider a heterogeneous MPSoC with at least two distinct processing units, which may use different instruction sets. The MPSoC additionally provides shared resources and communication between the processing units, including the possibility of issuing inter-processor interrupts from either processor and the availability of one or more memories that can be shared between the processors. We do not assume that any of the processors runs a full-fledged general purpose OS, such as Linux. Instead, all applications run either on an RTOS or bare metal. RTOS instances on each processing units may run in either an SMP configuration or with different instances on each core. This, in combination with the latency requirements described in the following, rules out the use of existing solutions such as OpenAMP. We do not consider any additional computational resources the MPSoC may provide, such as GPUs or FPGAs.

As mentioned in Section 1, a widely used hardware platform that matches these criteria is the Xilinx Zynq UltraScale+ MPSoC. Figure 1 shows the system setup described above, as mapped to this platform. To enable the different cores to communicate, we investigate both inter-processor interrupts and shared memory. For the latter, our target platform offers a number of different options, as well as a large configuration space, including various QoS mechanisms [35]. While we investigate the suitability of all memories or memory access configurations shown in Figure 2, we do not explore additional configuration options, apart from the use of the cache-coherent interconnect. We therefore end up with a total of four different memory configurations as options to build our solution on:

1. On-Chip Memory (OCM), with a total capacity of 256KiB, which is treated as non-cacheable memory by both processors.
2. DDR, using the default access paths for both processors, with a designated region of memory that is configured to be non-cacheable for both processors.
3. DDR, using the cache-coherent interconnect (CCI), with the real-time processor configured to use the coherent access path, and snooping of application processor caches from the CCI enabled (marked DDR/CCI in the following).

[1] `https://www.openampproject.org/`

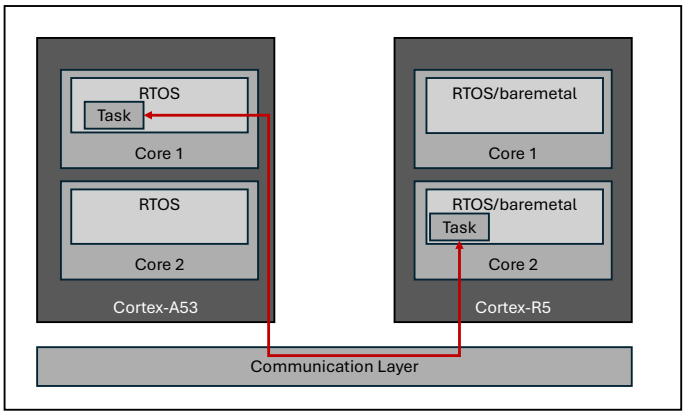


Figure 1 Our system setup as mapped to the processors of the Xilinx Zynq UltraScale+ platform, including our inter-processor communication layer for communicating between tasks on different processors.

4. Tightly Coupled Memory (TCM), with a total capacity of 128KiB for each real-time core, which is treated as non-cacheable memory by the application processor (and which is on the same level of the memory system as the caches for the real-time cores).

While only these shared memories are options for data transfer between processors, signaling can take place either over shared memory or using inter-processor interrupts ((5) in Figure 2).

The resulting solution is intended to be able to run low-latency real time applications, with response times below 30 microseconds, distributed across multiple processing units, including communication between processors in time-critical paths. Our primary aim is therefore to minimize latency, as we are only able to tolerate signaling and data transfer latencies between the processors of at most a few microseconds. Other considerations, such as reducing overall CPU utilization across the different processing units, are secondary.

We also aim to support two different kinds of communication between the processors in a distributed real-time task: Sporadic and periodic communication. The latter describes instances of communication that take place during every execution of a cyclical task - for instance, one processor sending an input for a computation to another and subsequently receiving its result. In this scenario, there is a corresponding *read* for each *write*, as in synchronous message passing, allowing for synchronization between tasks as well as data exchange. In sporadic communication, a task executing on one processor may occasionally update information that is used by a task running on the other processor, but no synchronization between the two occurs. It must be ensured that data read by the latter processor always remains internally consistent, and that its execution is not delayed by updates to the shared data (ruling out, for instance, any form of locking).

In the following, we describe our benchmarking and subsequent selection of hardware primitives to build our solution on in Section 3, and describe the solution and corresponding API in Section 4.

3 Benchmarking Results

In the following section, we will examine various metrics used to evaluate the proposed approach. These benchmarks were also instrumental in selecting the optimal memory type for fast and resilient control data transfer. All benchmarks were conducted on a Xilinx Zynq

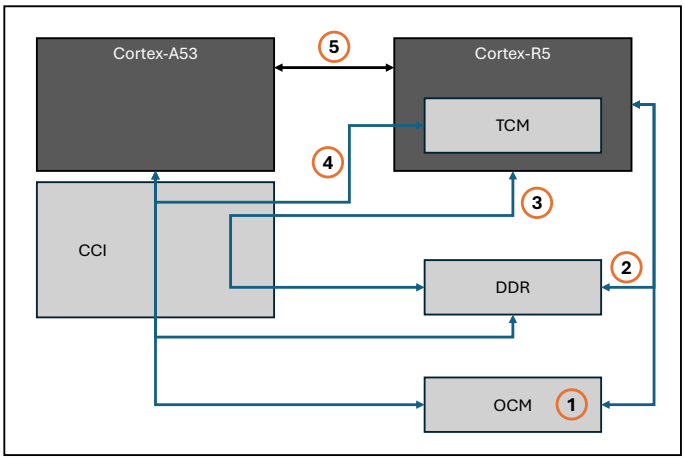


Figure 2 A simplified view of the different memory configurations and paths we consider for inter-processor communication.

UltraScale+ MPSoC, programmed using the AMD Vitis™ Unified Software Platform. The Ultra96 evaluation board used for testing is equipped with a Zynq UltraScale+ Ultra96 MPSoC, featuring a quad-core ARM Cortex-A53 CPU clocked at 1.2 GHz (the APU), a dual-core ARM Cortex-R5F CPU clocked at 500 MHz (the RPU), and 2 GiB of LPDDR4 memory.

The benchmarking scenario was structured as follows: Each test included 1,000 iterations, with 10 separate runs per test. To evaluate the impact of tasks running on another CPU core on inter-core communication performance, benchmarks were also conducted while a parallel FreeRTOS instance executed a memory-intensive task concurrently on another core of the APU. This memory-intensive task is based on a lightly modified version of the STREAM memory bandwidth benchmark[40], with changes to remove logging output and repeat benchmark runs indefinitely.

Time measurements for the benchmarks are taken on the target device itself. For measurements involving both processing units, this requires using one of the triple-timer counters (TTCs) available on the device. It is important to note that interacting with the TTCs itself impacts latency, which affects the measurements taken this way.

3.1 Shared Memory Latency Benchmarks

This section describes the benchmarks for shared memory latency in the configurations described in Section 2. We test unidirectional shared memory latencies between the processing units by starting a timer, using one of the TTCs, on one of the processors, and subsequently changing a single byte in the memory used in the given test scenario. The other processor continuously reads that byte and, as soon as the change becomes visible to it, stops the timer. These tests yield the results shown in Figure 3.

For the different memories and configurations (DDR, OCM, DDR with cache coherency, and TCM), the median latencies without additional load on the system are 1010 ns, 670 ns, 880ns, and 480 ns, respectively, going from the APU to the RPU. In the opposite direction, likewise without additional load, the median latencies for each memory type are 1000 ns, 490 ns, 840 ns, and 600 ns, respectively.

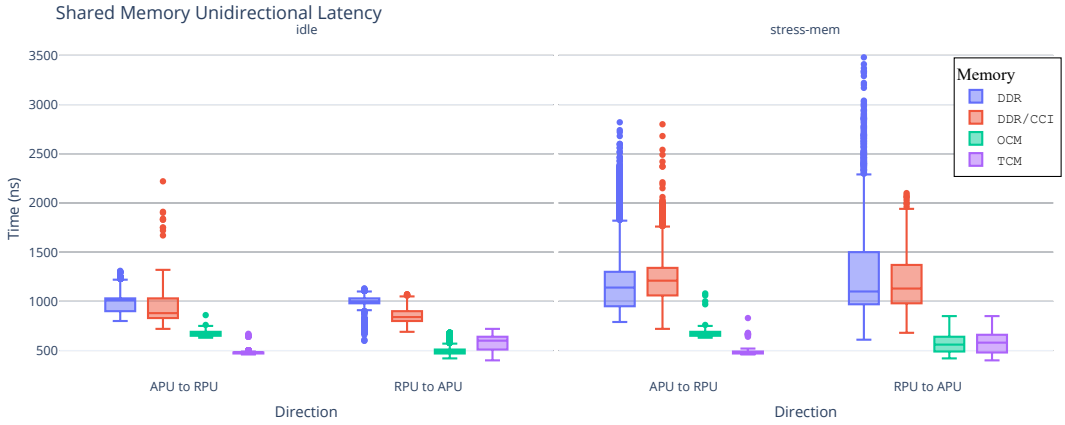


Figure 3 Shared memory unidirectional latency measured for four different memory types.

When a memory-intensive task is added on another APU core, the median latencies from the APU to the RPU increase to 1140 ns, 670 ns, 1210 ns, and 480 ns, respectively. In the opposite direction, the corresponding median latencies are 1100 ns, 560 ns, 1130 ns, and 580 ns, respectively. Additionally, there is a clear increase in the number of outliers in both directions when a memory-intensive task is added to the system. The effect is especially pronounced for the DDR and DDR/CCI cases. While the impact of the additional load could potentially be mitigated by adjusting QoS settings on the platform, we did not investigate these options as part of our testing.

We find that TCM and OCM show the fastest performance among the tested options, with TCM latencies being lower when going from APU to RPU, and OCM latencies being lower in the opposite direction. We also find these options are also the least sensitive to concurrent load. Finally, while the difference in latency between using TCM and OCM is quite small when going from RPU to APU, the latency advantage of the TCM in the opposite direction is more substantial, which suggests that the TCM is the most performant option overall.

3.2 Inter-Process Interrupt Latency

In this section, we describe the results of our benchmarks of inter-processor interrupt latency between the APU and the RPU. As in the shared memory latency benchmarks described in the previous section, we rely on the TTCs for our measurements: One core starts a timer and then immediately issues an inter-processor interrupt, at which point the other core will stop the timer in its interrupt handler. As previously, we perform these tests both on an idle system and with a memory-intensive task running on another APU core. The results of our benchmarks are shown in Figure 4.

With the other APU cores idle, we observe a median latency of 1970 ns going from the APU to the RPU and 1070 ns going in the other direction.

Adding our memory load does not substantially alter these values, however, it does result in a large increase in the number of outliers, especially when going from the APU to the RPU. This is significant as our solution is intended for low-latency real-time applications, where such increases in latency could lead to deadline misses and would not be tolerable.

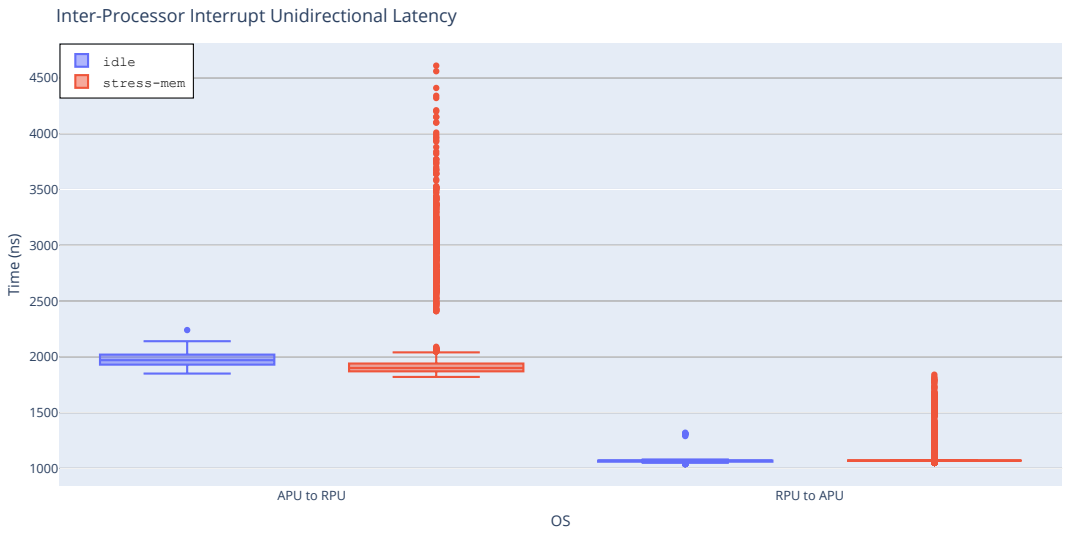


Figure 4 Inter-processor interrupt latencies.

Comparing these numbers to our previous results for shared memory latencies, we see that interrupt latencies are substantially higher in the median case, especially on an otherwise idle system. This holds true for all memories and configurations we tested, but the difference is especially pronounced when comparing the shared memory latency of the TCM to the inter-processor interrupt latencies we observed.

As outlined in Section 2, the primary goal of our solution is to optimize for latency. Based on these benchmark results, we therefore conclude that a solution relying only on shared memory for both signaling and data transfer is the most suitable as it allows for the lowest overall latency, even though this advantage comes at the cost of requiring busy-waiting on the receiving processing unit. We also conclude that, for scenarios where the amount of data is sufficiently small, and the required space in the TCM is available, the TCM should be the favored option, and we outline our solution building on this, and the performance for data transfer in the following.

3.2.1 Transferring Data via Tightly Coupled Memory (TCM)

Having chosen the TCM as the most performant option, we conduct additional testing to determine how long it takes to transfer data from the APU to the RPU in this way. While our previous tests only wrote or read a single byte, this test scenario involves one processor writing an array of varying size to the shared memory region, setting a flag to indicate that it is ready to be read, and the other processor reading the full array. The results of these tests are shown in Figure 5. Adding a memory load to the system increases data transfer latency substantially, as well as leading to an increasing number of outliers. Overall latency grows quickly for increasing data transfer sizes, showing that, while communication between processors is feasible even on time-critical paths, it is crucial to minimize the amount of data that needs to be sent or received. While up to 128 B have a limited impact, latency increases quickly after that point.

These results form the foundation of the inter-processor communication solution we describe in the following section.

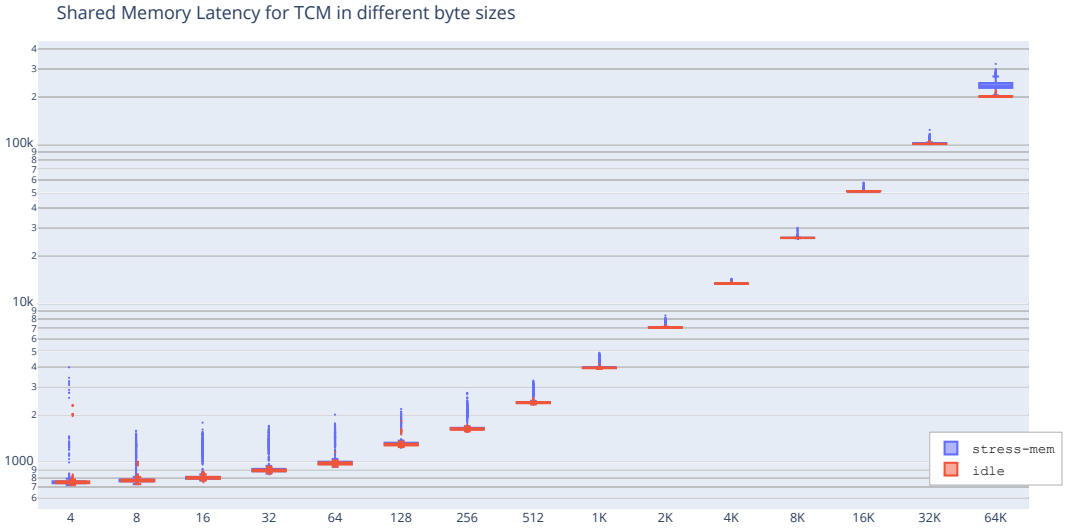


Figure 5 Latency for transferring data from the APU to the RPU via the TCM memory.

4 Inter-Core Communication Solution

As our benchmark results show, relying only on shared memory for signaling can achieve lower latencies than inter-processor interrupts. This comes at the cost of requiring the receiving side to busy-wait, thereby hurting overall efficiency since the receiving side is not able to perform useful work while waiting, and increasing CPU utilization. Nonetheless, we choose to build our solution only on shared memory, following the priorities described in Section 2.

As previously discussed in the same Section, we distinguish between two different types of inter-core communication: sporadic and periodic communication. We define periodic communication as communication that takes place during every execution of a cyclical task, for example, one processor sending an input for a computation to another and subsequently receiving its result. On the contrary, sporadic communication occurs only occasionally, for example, a task executing on one processor may update information that is used by a task running on another processor without any synchronization.

To facilitate an inter-core communication (ICC) between processors for these two communication types, we implement two concepts commonly used for inter-process communication: A mailbox and a double buffer. Figure 6 outlines the proposed inter-core communication API and its individual components.

The periodic communication is realized via a shared memory mailbox: A single-reader, single-writer mailbox that supports alternating reads and writes, i.e., for each write there must be a corresponding read prior to the next write. An in depth description is given in Section 4.1.

The sporadic communication is realized via a shared memory double buffer: A single-writer, multiple-reader buffer that may be read multiple times without restriction and can be updated independently of the reading process, provided that the update frequency remains lower than the read frequency. A detailed explanation is given in Section 4.2.

The ICC API is implemented as a header-only `C++` library, without any platform- or hardware-specific code. Users need to provide a shared memory region visible to both communicating cores and instantiate the shared memory structures on both sides. For

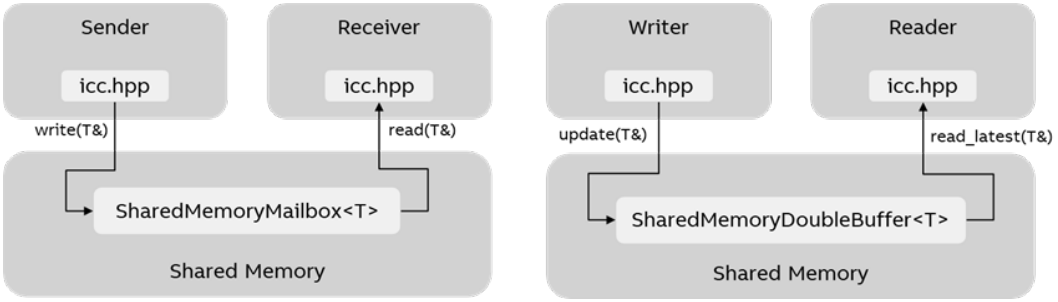
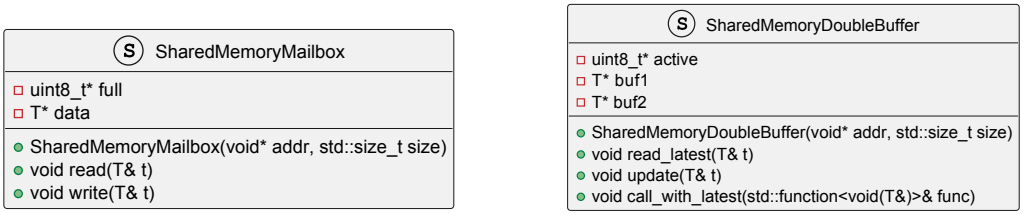


Figure 6 Overview of the proposed inter-core communication API.



(a) Class diagram of the shared memory mailbox. **(b)** Class diagram of the double buffer.

Figure 7 The proposed data structures for inter-processor communication.

performance reasons, all data structures shared over this API should be trivially copyable and share the same layout on both sides. While adding serialization support is possible, and we have investigated a lightweight serialization extension to this solution, we omit a detailed discussion of this topic.

4.1 Shared Memory Mailbox

For cyclic communication, we propose to use a mailbox in shared memory. A slice of memory is allocated within a shared memory region, which is accessible to all participants. It serves as both a buffer, allowing messages to be written and read by various entities, and, by way of a status flag, a synchronization mechanism. The mailbox interface governs the interactions between the shared memory mailbox and the entities using it.

Our implementation of the shared memory mailbox uses a templated structure that is designed to store and transfer a single instance of a data type `<T>`, e.g., an integer value, in a shared memory segment. By employing `C++` templates, we ensure that the structure is capable of handling any data type required across various use cases. Figure 7a provides an overview of the shared memory mailbox, its variables, and functions. The `data` variable holds a pointer to the actual data of type `<T>` that will be exchanged via the mailbox. The `full` variable points to a flag which indicates whether the mailbox contains data. The constructor assigns addresses in the shared memory region to both of these, as well as ensuring that the shared memory region is sufficiently large and that `T` is trivially copyable. Note that `T` must have the same layout for all users of the mailbox, and this is not checked automatically.

To read data from the mailbox, readers can use the `read` function, providing a reference to an instance of `T` that will be updated with the data read from the mailbox. `read` is blocking, and is implemented using busy-waiting. If the mailbox is empty, the reader loops until the `full` flag is set and new data is available to read. After reading, the reader resets

the flag to indicate that the mailbox is empty, allowing a writer to place new data there. Atomic operations are used to ensure sequential consistency and prevent data races or issues due to reordering.

To write data to the mailbox, writers can use the `write` function, providing a reference to an instance of `T` that will be copied to the mailbox. Like `read`, `write` is blocking: If the mailbox is already full, the writer loops until it is empty and new data can be placed there.

Our shared memory mailbox is a lightweight structure that allows communication between tasks on different processors with a simple, blocking API. It enables both data exchange and synchronization, and would be straightforward to extend with a non-blocking API (e.g., `bool try_write(T& t)`, `bool try_read(T& t)`).

4.2 Shared Memory Double-Buffer

For sporadic communication, we propose to use a double buffer placed in shared memory. Like our shared memory mailbox, it only requires a slice of shared memory accessible to all participants. The double buffering technique is a widely used approach to ensure data consistency when reading and writing data concurrently in a shared memory environment. It ensures that the reader gets a complete and consistent view of the data, even if the writer is currently updating it. Figure 7b provides an overview of the shared memory double buffer, its variables, and functions. The shared memory double buffer contains a pointer`active` that indicates which of the two buffers is currently active, i.e., should be used for reading, and two buffers `buf1` and `buf2`, which are pointers to two instances of the type `<T>`. As for the mailbox, the constructor assigns addresses in the shared memory region to these pointers, checks that the region is sufficiently large, and that `T` is trivially copyable. As before, users must ensure that `T` additionally shares the same layout for all participants.

Readers can use the `read_latest` function to read the latest value from the double buffer. Unlike the mailbox's `read`, this is non-blocking, and will immediately update the provided instance of `T`. Reading does not change `active`, and the same value can be read multiple times.

To update the double buffer, writers can use the `update` function. This function checks the value of `active`, selects the currently inactive buffer, and overwrites it with the contents of the provided `T`, before updating `active` to point to the newly written value. Like `read_latest`, `update` is non-blocking. As in the mailbox, atomic operations are used to ensure sequential consistency.

It is important to note that this implementation allows for data races if updates happen at a higher frequency than reads: If a reader calls `read_latest`, and the writer calls `update` more than once while the reader is reading, it will affect the buffer used by the reader, which may lead to inconsistent data being read. In practice, this issue can be avoided by ensuring that the writing task τ_w has a longer task period than any reading task τ_r^i, and only updates the double buffer once per period. If this cannot be guaranteed, additional synchronization would be required, which may in turn lead to higher delays.

5 Related Work

In the rapidly evolving domain of Multi-Processor Systems-on-Chip (MPSoCs), extensive research has been conducted to enhance performance verification, communication architectures, task scheduling, and memory management. This section reviews key contributions, providing a comprehensive context for the solution proposed in this paper.

Inter-processor communication is a critical aspect of MPSoC design. Hung et al. [20] introduced MSG, an inter-core communication mechanism derived from MPI and MCAPI, showing improved performance on platforms like IBM CELL. Schwäricke et al. [34] used cache partitioning in a real-time communication framework, which is applicable to managing memory contention. These studies emphasize established protocols and cache management, yet our research distinguishes itself by using shared memory exclusively for signaling and data exchange, avoiding inter-processor interrupts to achieve sub-microsecond latencies essential for time-critical applications. Loghi et al. [25] provided insights into communication bottlenecks using SystemC simulations, while Nieto et al. [28] explored synchronization methods on the Zynq-UltraScale+, comparing techniques like InterProcessor Interrupts (IPIs) with shared memory. Our work builds on these by demonstrating that removing IPIs in favor of shared memory not only simplifies the communication model but also enhances latency performance, pivotal for real-time applications across heterogeneous units. Further studies by Mirosanlou et al. [26], Belloch et al. [5], Hassan et al. [18], and Nollet et al. [29] investigated communication strategies and low-latency interconnects relevant to shared memory systems.

Performance verification is crucial for MPSoCs. Richter et al. [32] and Castells-Rufas et al. [8] provided foundational methodologies for assessing communication performance. France-Pillois et al. [11] and Shanthi et al. [36] developed tools to optimize synchronization barriers, enhancing communication efficiency in real-time systems. Our study offers a streamlined solution with concrete benchmarking results, demonstrating the practical application of shared memory in achieving low-latency communication without traditional synchronization complexities.

Effective memory management underpins low-latency communication. Bansal et al. [4] and Carletti et al. [7] tackled cache coherency and memory contention, proposing techniques like cache coloring. Our approach simplifies memory access patterns by using shared memory, reducing contention and bypassing cache coherency challenges, thus enhancing predictability and efficiency. Hoornaert et al. [19], Hahn et al. [17], Kogel et al. [22], and Acacio et al. [1] focused on memory subsystem optimizations and architectural strategies for efficient memory access.

Optimizing task scheduling is vital for communication efficiency. Singh et al. [37] and Ali et al. [2] developed heuristics and methodologies for runtime mapping and energy efficiency. Spieck et al. [39] emphasized minimizing communication overhead. Our approach prioritizes achieving minimal communication latency over resource utilization, dedicating one processor to this task, aligning with the stringent requirements of real-time applications. Ristau et al. [33] and Guerin et al. [15] explored strategies that optimize resource usage and reduce communication latency, critical for efficient shared memory communication.

Real-time applications often operate within mixed criticality systems, which require predictable performance across varying criticality levels. Gracioli et al. [14] proposed frameworks that use dedicated memory interfaces and cache coloring to ensure timing predictability. Nowotsch et al. [30] focused on managing resource contention to maintain timing predictability. Our work uniquely addresses the latency needs of these applications through a shared memory-based model that avoids additional complexities, ensuring timing predictability in mixed criticality environments. Geyer et al. [13] and Kim et al. [21] provide additional context for managing real-time constraints.

Several studies, while not directly focused on shared memory communication, provide valuable insights into MPSoC design and optimization. Research on virtualization in mixed criticality systems [10, 9, 31], as well as security and energy efficiency in IoT applications [42, 16], highlight the importance of dynamic resource allocation and energy efficiency,

complementing shared memory communication strategies. Advanced resource management techniques explored by Kumar et al. [23] and Frid et al. [12] contribute to optimizing performance and energy efficiency.

Further contributions include studies on programming models and software optimization, such as Sommer et al. [38], which examines parallel programming models for automotive applications, and Brown et al. [6], who introduced the Distributed Multiloop Language (DMLL) to optimize parallel applications. Our approach offers an easy-to-use, header-only C++ library that facilitates porting across different platforms, emphasizing practical applicability in real-time environments. Liu et al. [24] explores flexible inter-core communication methods in multicore processors, demonstrating low latency and high system performance, aligning with the objectives of optimizing shared memory communication. The work by Tsao et al. [41] and Nie et al. [27] on performance evaluation and design of multiprocessor embedded systems using shared memory further supports the development of efficient communication solutions. Anjum et al. [3] provides insights into the latest methodologies for optimizing inter-processor communication.

6 Conclusion

Deploying low-latency real-time applications across heterogeneous processors on modern MPSoCs poses a challenging problem, especially if communication between processors is required in a time-critical path.

We present benchmarking results and a proof-of-concept solution showing that, by avoiding inter-processor interrupts, relying only on shared memory for signaling and data exchange, and sacrificing efficient utilization of one of the processors to minimize latency, we can achieve signaling latencies and data transfer times in the low microsecond range on the Xilinx Zynq UltraScale+ platform.

We implement our solution in an easy-to-use, header-only C++ library. While our solution and implementation are straightforward to port to other hardware platforms, making the most effective use of it on another platform would entail additional benchmarking to select a suitable memory configuration.

References

1 Manuel E Acacio, Jose Gonzalez, Jose M Garcia, and Jose Duato. An architecture for high-performance scalable shared-memory multiprocessors exploiting on-chip integration. *IEEE Transactions on Parallel and Distributed Systems*, 15(8):755–768, 2004. doi:10.1109/TPDS.2004.27.

2 Haider Ali, Xiaojun Zhai, Umair Ullah Tariq, and Lu Liu. Energy efficient heuristic algorithm for task mapping on shared-memory heterogeneous mpsocs. In *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 1099–1104. IEEE, 2018. doi:10.1109/HPCC/SMARTCITY/DSS.2018.00183.

3 Etsam Anjum and Jeffrey Hancock. Introduction to openamp library. https://www.openampproject.org/docs/whitepapers/Introduction_to_OpenAMPlib_v1.1a.pdf, 2023. Accessed: October 29, 2024.

4 Ayoosh Bansal, Rohan Tabish, Giovani Gracioli, Renato Mancuso, Rodolfo Pellizzoni, and Marco Caccamo. Evaluating memory subsystem of configurable heterogeneous mpsoc. *Proceedings of the Operating Systems Platforms for Embedded Real-Time applications*, 2018.

5 Jose A Belloch, Germán León, José M Badía, Almudena Lindoso, and Enrique San Millan. Evaluating the computational performance of the xilinx ultrascale+ eg heterogeneous mpsoc. *The Journal of Supercomputing*, 77:2124–2137, 2021. doi:10.1007/S11227-020-03342-7.

6 Kevin J Brown, HyoukJoong Lee, Tiark Rompf, Arvind K Sujeeth, Christopher De Sa, Christopher Aberger, and Kunle Olukotun. Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 194–205, 2016. doi:10.1145/2854038.2854042.

7 Lorenzo Carletti, Gianluca Brilli, Alessandro Capotondi, Paolo Valente, and Andrea Marongiu. The importance of worst-case memory contention analysis for heterogeneous socs. *arXiv preprint*, 2023. doi:10.48550/arXiv.2309.12864.

8 David Castells-Rufas, Jaume Joven, Sergi Risueño, Eduard Fernandez, Jordi Carrabina, Thomas William, and Hartmut Mix. Mpsoc performance analysis with virtual prototyping platforms. In *2010 39th International Conference on Parallel Processing Workshops*, pages 154–160. IEEE, 2010.

9 Marcello Cinque, Domenico Cotroneo, Luigi De Simone, and Stefano Rosiello. Virtualizing mixed-criticality systems: A survey on industrial trends and issues. *Future Generation Computer Systems*, 129:315–330, 2022. doi:10.1016/J.FUTURE.2021.12.002.

10 Marcello Cinque, Gianmaria De Tommasi, Sara Dubbioso, and Daniele Ottaviano. Rpuguard: Real-time processing unit virtualization for mixed-criticality applications. In *2022 18th European Dependable Computing Conference (EDCC)*, pages 97–104. IEEE, 2022. doi:10.1109/EDCC57035.2022.00025.

11 Maxime France-Pillois, Jérôme Martin, and Frédéric Rousseau. A non-intrusive tool chain to optimize mpsoc end-to-end systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 18(2):1–22, 2021. doi:10.1145/3445030.

12 Nikolina Frid, Danko Ivošević, and Vlado Sruk. Performance estimation in heterogeneous mpsoc based on elementary operation cost. In *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1202–1205. IEEE, 2016.

13 Tobias Geyer, Nikolaos Oikonomou, Georgios Papafotiou, and Frederick D Kieferndorf. Model predictive pulse pattern control. *IEEE Transactions on Industry Applications*, 48(2):663–676, 2011.

14 Giovani Gracioli, Rohan Tabish, Renato Mancuso, Reza Mirosanlou, Rodolfo Pellizzoni, and Marco Caccamo. Designing mixed criticality applications on modern heterogeneous mpsoc platforms. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019.

15 Xavier Guerin, Katalin Popovici, Wassim Youssef, Frederic Rousseau, and Ahmed Jerraya. Flexible application software generation for heterogeneous multi-processor system-on-chip. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, volume 1, pages 279–286. IEEE, 2007.

16 Marisabel Guevara, Benjamin Lubin, and Benjamin C Lee. Navigating heterogeneous processors with market mechanisms. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 95–106. IEEE, 2013. doi:10.1109/HPCA.2013.6522310.

17 Sebastian Hahn, Michael Jacobs, and Jan Reineke. Enabling compositionality for multicore timing analysis. In *Proceedings of the 24th international conference on real-time networks and systems*, pages 299–308, 2016. doi:10.1145/2997465.2997471.

18 Mohamed Hassan. Heterogeneous mpsocs for mixed criticality systems: Challenges and opportunities. *arXiv preprint*, 2017. arXiv:1706.07429.

19 Denis Hoornaert, Shahin Roozkhosh, and Renato Mancuso. A memory scheduling infrastructure for multi-core systems with re-programmable logic. In *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

20 Shih-Hao Hung, Wen-Long Yang, and Chia-Heng Tu. Designing and implementing a portable, efficient inter-core communication scheme for embedded multicore platforms. In *2010 IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 303–308. IEEE, 2010. doi:10.1109/RTCSA.2010.17.

21 Hyoseung Kim, Dionisio De Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–154. IEEE, 2014. doi:10.1109/RTAS.2014.6925998.

22 Tim Kogel and Heinrich Meyr. Heterogeneous mp-soc: the solution to energy-efficient signal processing. In *Proceedings of the 41st annual Design Automation Conference*, pages 686–691, 2004. doi:10.1145/996566.996754.

23 Akash Kumar, Bart Mesman, Bart Theelen, Henk Corporaal, and Yajun Ha. Analyzing composability of applications on mpsoc platforms. *Journal of Systems Architecture*, 54(3-4):369–383, 2008. doi:10.1016/J.SYSARC.2007.10.002.

24 Cheng Liu, Lei Luo, Meng Li, Pinyuan Lei, Lirong Chen, and Kun Xiao. Inter-core communication mechanisms for microkernel operating system based on signal transmission and shared memory. In *2021 7th International Symposium on System and Software Reliability (ISSSR)*, pages 188–197. IEEE, 2021.

25 Mirko Loghi, Federico Angiolini, Davide Bertozzi, Luca Benini, and Roberto Zafalon. Analyzing on-chip communication in a mpsoc environment. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, volume 2, pages 752–757. IEEE, 2004. doi:10.1109/DATE.2004.1268966.

26 Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. Duetto: Latency guarantees at minimal performance cost. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1136–1141. IEEE, 2021. doi:10.23919/DATE51398.2021.9474062.

27 Yang Nie, Lili Jing, and Pengyu Zhao. Design and implementation of inter-core communication of embedded multiprocessor based on shared memory. *International Journal of Security and Its Applications*, 10(12):21–30, 2016.

28 Rubén Nieto, Edel Díaz, Raúl Mateos, and Álvaro Hernández. Evaluation of software interprocessor synchronization methods for the zynq-ultrascale+ architecture. In *2020 XXXV Conference on Design of Circuits and Integrated Systems (DCIS)*, pages 1–6. IEEE, 2020. doi:10.1109/DCIS51330.2020.9268616.

29 Vincent Nollet, Prabhat Avasare, J-Y Mignolet, and Diederik Verkest. Low cost task migration initiation in a heterogeneous mp-soc. In *Design, Automation and Test in Europe*, pages 252–253. IEEE, 2005.

30 Jan Nowotsch and Michael Paulitsch. Leveraging multi-core computing architectures in avionics. In *2012 Ninth European Dependable Computing Conference*, pages 132–143. IEEE, 2012. doi:10.1109/EDCC.2012.27.

31 D Ottaviano, M Cinque, G Manduchi, and S Dubbioso. Virtualization of accelerators in embedded systems for mixed-criticality: Rpu exploitation for fusion diagnostics and control. *Fusion Engineering and Design*, 190:113518, 2023.

32 Kai Richter, Marek Jersak, and Rolf Ernst. A formal approach to mpsoc performance verification. *Computer*, 36(4):60–67, 2003. doi:10.1109/MC.2003.1193230.

33 Bastian Ristau and Gerhard Fettweis. Mapping and performance evaluation for heterogeneous mp-socs via packing. In *Embedded Computer Systems: Architectures, Modeling, and Simulation: 7th International Workshop, SAMOS 2007, Samos, Greece, July 16-19, 2007. Proceedings 7*, pages 117–126. Springer, 2007. doi:10.1007/978-3-540-73625-7_14.

34 Gero Schwäricke, Rohan Tabish, Rodolfo Pellizzoni, Renato Mancuso, Andrea Bastoni, Alexander Zuepke, and Marco Caccamo. A real-time virtio-based framework for predictable inter-vm communication. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 27–40. IEEE, 2021. doi:10.1109/RTSS52674.2021.00015.

35 Alejandro Serrano Cases, Juan M Reina, Jaume Abella Ferrer, Enrico Mezzetti, and Francisco Javier Cazorla Almeida. Leveraging hardware qos to control contention in the xilinx zynq ultrascale+ mpsoc. In *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021): July 5-9, 2021, Virtual Conference*, volume 196, pages 3–1. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ECRTS.2021.3.

36 D Shanthi and R Amutha. Performance analysis of on-chip communication architecture in mpsoc. In *2011 International Conference on Emerging Trends in Electrical and Computer Technology*, pages 811–815. IEEE, 2011.

37 Amit Kumar Singh, Wu Jigang, Alok Prakash, and Thambipillai Srikanthan. Efficient heuristics for minimizing communication overhead in noc-based heterogeneous mpsoc platforms. In *2009 IEEE/IFIP International Symposium on Rapid System Prototyping*, pages 55–60. IEEE, 2009. doi:10.1109/RSP.2009.18.

38 Lukas Sommer, Florian Stock, Leonardo Solis-Vasquez, and Andreas Koch. Using parallel programming models for automotive workloads on heterogeneous systems-a case study. In *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 17–21. IEEE, 2020. doi:10.1109/PDP50117.2020.00010.

39 Jan Spieck, Stefan Wildermann, and Jürgen Teich. A learning-based methodology for scenario-aware mapping of soft real-time applications onto heterogeneous mpsocs. *ACM Transactions on Design Automation of Electronic Systems*, 28(1):1–40, 2022. doi:10.1145/3529230.

40 STREAM: Sustainable Memory Bandwidth in High Performance Computers. https://www.cs.virginia.edu/stream/. Accessed: 2024-11-12.

41 Shiao-Li Tsao, Sung-Yuan Lee, et al. Performance evaluation of inter-processor communication for an embedded heterogeneous multi-core processor. *Journal of information science and engineering*, 28(3), 2012. URL: http://www.iis.sinica.edu.tw/page/jise/2012/201205_07.html.

42 Junlong Zhou, Jin Sun, Peijin Cong, Zhe Liu, Xiumin Zhou, Tongquan Wei, and Shiyan Hu. Security-critical energy-aware task scheduling for heterogeneous real-time mpsocs in iot. *IEEE Transactions on Services Computing*, 13(4):745–758, 2019. doi:10.1109/TSC.2019.2963301.

Co-Design of Systems-On-Chip for Sustainability

Jan Spieck
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Dominik Walter
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Jan Waschkeit
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Jürgen Teich
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Abstract

This paper introduces a novel approach to the co-design of sustainable embedded systems through multi-objective design space exploration (DSE). We propose a two-phase methodology that optimizes both the multiprocessor system-on-chip (MPSoC) architecture and application mappings, considering sustainability, reliability, performance, and cost as optimization objectives of equal importance. Unlike existing approaches, our method thereby accounts for both operational and embodied emissions, providing a more comprehensive assessment of sustainability. The first phase employs intra-application DSEs to explore Pareto-optimal constraint graphs for each application. The second phase, an inter-application DSE, combines these results to explore sustainable target architectures and corresponding application mappings. Our approach incorporates detailed models for embodied emissions (scope 1 and scope 2), operational emissions, reliability, performance, and cost. The evaluation demonstrates that our sustainability-aware DSE is able to explore design spaces overlooked by traditional approaches, supported by superior results in four key objectives. This enables the development of more environmentally friendly embedded systems while still achieving high performance and reliability.

2012 ACM Subject Classification Computer systems organization → Embedded and cyber-physical systems; Hardware → Power and energy; Hardware → Chip-level power issues; Computer systems organization → System on a chip; Social and professional topics → Sustainability; Hardware → Impact on the environment

Keywords and phrases System-on-Chip, Sustainability, Multi-objective Optimization, Design Space Exploration, Embedded Systems, Carbon Emissions

Digital Object Identifier 10.4230/OASIcs.NG-RES.2025.3

Funding *Jan Spieck*: Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project number 146371743 – TRR 89: Invasive Computing
Dominik Walter: Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project number 146371743 – TRR 89: Invasive Computing

1 Introduction

The impact of information and communication technology (ICT) on global greenhouse gases (GHGs) emissions is currently estimated to be between 2.1 % and 3.9 %, and is accelerating rapidly [11]. But since global emissions must be reduced significantly over the next years, a fast shift to more sustainable computing systems is inevitable. However, sustainability does not come for free and to design a sustainable system that balances performance, reliability, and at the same time cost poses a complex challenge. This is to our knowledge the first work to provide a solution to this problem by proposing a highly configurable multi-objective sustainability-aware design space exploration (DSE) that explores sustainable embedded system designs automatically. More specifically:

1. We propose a DSE for exploring sustainable MPSoC target architectures, optimizing for sustainability, reliability, performance and cost. It can be fine-tuned to the specific production and deployment environment of the embedded system to accurately model the CO_2 emissions during both production and operation.
2. Besides exploring sustainable target architectures, the proposed approach explores also composable application mapping for real-time applications, that are tailored to these architectures. Besides applications with static execution properties, the approach even supports applications with dynamic workload variations during operation.
3. A comprehensive evaluation of the proposed sustainability-aware DSE is provided to demonstrate the need to consider sustainability during the design.

2 Related Work

Design space exploration (DSE) has become a de-facto standard for simultaneous optimization of various aspects of multiprocessor systems-on-chip (MPSoC) architectures. However, existing approaches overlook the significant impact of the embodied emissions of an embedded system, although [9] predicts that these embodied emissions will dominate the total emissions for future embedded systems. [27] propose a scenario-aware DSE that explores a MPSoC architecture and corresponding scenario-aware mappings using a co-evolutionary algorithm concerning the objectives of execution time and chip area. However, their DSE has scalability issues for large number of applications and big architectures. [1] present a DSE that optimizes the mapping of applications regarding execution time, lifetime, power, and temperature for a given target architecture. Furthermore, it turns the multi-objective optimization problem into a single-objective one, thus losing large parts of the search space. [19] provide an overview of hybrid application mapping techniques for heterogeneous MPSoC target architectures, including approaches for DSE. Instead of a DSE, also heuristic mapping approaches can be used at run time to speedup the exploration [22]. [21] propose a symmetry-eliminating DSE, enabling efficient exploration of so-called constraint graphs that specify an application mapping on a higher abstraction level. By doing so, the exploration can skip the vast inherent redundancy of mappings. [9] and [12] present a detailed analysis of the environmental impact of computing systems, considering both operational and embodied emissions. Their work highlights the importance of a holistic approach to sustainability in computing. [25] present a tool to analyze the carbon footprint of chiplet-based architectures, showing how it can reduce greenhouse gas emissions in manufacturing. [5] propose a co-optimization approach for datacenter sustainability, considering both design technology and system-level factors, thereby balancing performance and sustainability objectives.

3 Fundamentals

In this section, we introduce our application and architecture model, describe the corresponding formal mapping and give a short overview on modeling and evaluating of sustainability and reliability, using the illustrative example in Fig. 1.

An application $a \in A$ is a directed graph of tasks $\tau \in T_a$, communicating over channels. These graphs express data dependencies between the tasks by direct edges; a task can start when the preceding tasks have finished processing. At run time, applications process data $d \in D_a$ periodically, where the processing of each data should finish before an application-specific deadline θ_a. The task graphs of three example applications $a_1, a_2, a_3 \in A$ are depicted at the top of Fig. 1.

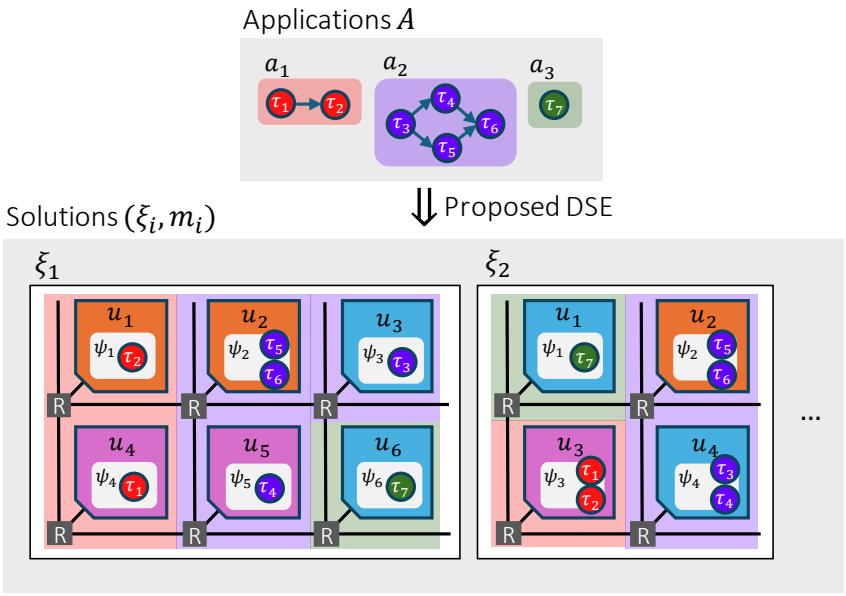


Figure 1 Illustration of application graphs for three example applications $a_1, a_2, a_3 \in A$ and two target MPSoC solutions ξ_1, ξ_2 with two annotated mappings, explored by the proposed DSE. Architecture ξ_1 is equipped with six tiles of three different types, while ξ_2 has four tiles. These tiles are connected by a 2D NoC mesh using routers. Each tile is equipped with a single-core processor of different types, onto which the tasks of the applications are mapped.

This work explores heterogeneous tile-based MPSoC architectures ξ, as presented by [3, 7, 14, 6]. Each tile $u \in U$ is equipped with a set of processing resources $\psi \in u$, e.g., CPUs or accelerators [28], connected to a 2D Network-on-Chip (NoC) mesh, and is of a heterogeneous tile type $v \in V$, differing in the number and type of processors on the tile. Two example target architectures ξ_1 and ξ_2 are depicted in the bottom row of Fig. 1. The MPSoC ξ_1 is equipped with six tiles $u_1, \ldots, u_6$ of three different color-coded tile types v_1, v_2, v_3, defined by the type of the mounted single-core processor ψ_i. ξ_2 contains four tiles $u_1, \ldots, u_4$.

A mapping m maps each application task $\tau \in T$ to exactly one resource ψ of the target architecture ξ. Different applications are always mapped to separate sets of tiles of the target architecture, i.e., there are no tiles with tasks from two different applications. This allows composability [2] which is achieved by applying a temporal isolation scheme [13] to the NoC. As the search space for mappings is polluted by many redundant mappings with a different binding but the same execution characteristics, [21] proposes a symmetry-eliminating DSE. Instead of exploring individual mappings m directly, this approach explores *constraint graphs* $\eta \in H$, which group tasks together, assign these groups to tile types, and specify constraints between tasks, e.g., a maximum hop distance on a NoC. This abstraction captures essential mapping characteristics without tying to a specific chip layout, thus eliminating redundant solutions. As a result, the proposed DSE efficiently considers only meaningfully design alternatives, later transforming these graphs into actual task-to-resource mappings m by a binding algorithm [23]. Fig. 1 illustrates the transformed mappings for the three applications to the two target architectures. On ξ_1, the application a_1 with tasks $\tau_1, \tau_2 \in T_{a_1}$ is mapped to the tiles $u_1, u_4 \in \xi_1$, the application a_2 with tasks $\tau_3, \ldots, \tau_6 \in T_{a_2}$ is mapped to the tiles $u_2, u_3, u_5 \in \xi_1$, and the application a_3 with tasks $\tau_7 \in T_{a_3}$ is mapped to tile $u_6 \in \xi_1$. This mapping also includes access grants to the routers of the NoC, which are necessary for inter-tile communication. Another mapping is illustrated for ξ_2.

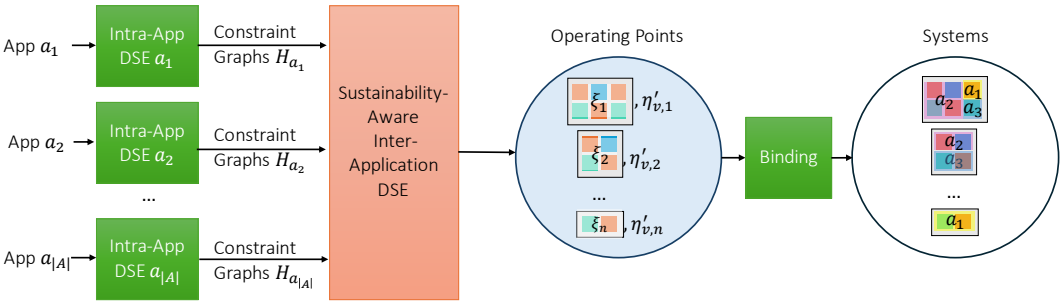


Figure 2 Optimization flow of the proposed sustainability-aware DSE of embedded SoC systems that implement a given set A of applications.

In this work, we analyze and systematically explore the design space of tile-based multiprocessor system-on-chip (MPSoC) implementations of embedded systems with respect to sustainability by considering both their so-called *embodied* and *operational* greenhouse gas (GHG) emissions. *Operational* emissions are the emissions that occur during the operation of the system, while the *embodied* emissions refer to the emissions that occur during production of the chip. These can be differentiated into three *scopes* [29]. *Embodied Scope 1* emissions include chemicals and gases produced directly during chip manufacturing. The emissions are represented in CO_2 equivalents. *Embodied Scope 2* emissions represent the portion of emissions attributed to the energy required during the manufacturing process. Unlike scope 1 emissions which occur directly within the production environment, scope 2 emissions are dependent on the energy mix used to power the production facility. This energy mix varies by location based on the region's reliance on renewable or fossil fuel energy sources. *Embodied Scope 3* emissions include upstream and downstream supply-chain emissions and are omitted in this work due to their high variability and difficulty in estimation.

While operational emissions can be reduced by designing more energy-efficient computer architectures, embodied emissions can only be reduced – from a hardware design perspective – by either building smaller less powerful chips or extending the lifetime of the chip to amortize the fixed emissions during manufacturing. Therefore, the lifetime, or more generally the reliability, of a system plays a critical role in achieving a sustainable design.

4 DSE for sustainable Co-Design

Exploring a sustainable MPSoC architecture with corresponding application mappings poses a complex multi-objective optimization problem. In this work, we propose to employ the two-phase exploration methodology displayed in Fig. 2 to solve this optimization problem. For a given set A of applications to be implemented on the system to be designed, we first perform a set of separate intra-application DSEs for each application $a \in A$ to explore Pareto-optimal *constraint graphs* $\eta_a \in H_a$ for a variety of tile number combinations using [21]. Afterwards, an inter-application DSE explores the space of architectures ξ and constraint graphs $\eta_v = \{\eta_{a_1}, ..., \eta_{a_{|A|}}\}$, maximizing sustainability, performance, and reliability while minimizing cost. Finally, the constraint graphs are bound to the target architecture using the binding algorithm from [23], resulting in a set of target designs with varying application mappings.

4.1 Intra-Application DSE

For each application $a \in A$, a separate intra-application DSE explores so-called *constraint graphs* $\eta_a \in H_a$, which minimize the deadline miss percentage $dm(\eta_a, \xi)$ and energy consumption $eng(\eta_a, \xi)$ when executing on an architecture ξ, constituted by the required number of tiles $r_{v_i}(\eta_a)$ of each tile type $v_i \in V$ for the regarded constraint graph η_a:

$$\underset{\eta_a \in H_a}{\text{minimize}}(dm(\eta_a, \xi), eng(\eta_a, \xi), r_{v_1}(\eta_a), ..., r_{v_{|V|}}(\eta_a)) \tag{1}$$

For applications with constant execution times, $dm(\eta_a, \xi)$ is binary, being 0 if the execution time $l(\eta_a, \xi)$ of the application a under a representative mapping for constraint graph η_a on ξ is smaller than the deadline θ_a and 1 otherwise. For applications with input-dependent execution, the deadline miss percentage can be determined by a representative dataset D:

$$dm(\eta_a, \xi) = \frac{1}{|D|} \cdot \sum_{d \in D} \begin{cases} 0 & \text{if } l(\eta_a, \xi, d) \leq \theta_a \\ 1 & \text{else} \end{cases} \tag{2}$$

Let eng represent the total energy consumption of the chip during its lifetime, which we calculate by the average energy consumption eng_{θ_a} per period (equalling the deadline θ), scaled up to its lifetime ϑ:

$$eng(\eta_a, \xi) = \phi \cdot \frac{\vartheta}{\theta_a} \cdot eng_{\theta_a}(\eta_a, \xi) \tag{3}$$

In this formula, ϕ denotes the utilization of the MPSoC, expressing the percentage the SoC is active during its lifetime. $eng_{\theta_a}(\eta_a, \xi)$ is then given as the sum of the energy consumption per task $\tau \in T_a$, which is the product of the power consumption $P(\psi)$ of a compute resource $\psi = \eta_a(\tau)$ and the average latency of the task τ:

$$eng_{\theta_a}(\eta_a, \xi) = \sum_{\tau \in T_a} \left(P(\eta_a(\tau)) \cdot \frac{1}{|D|} \sum_{d \in D_a} l_\tau(\eta_a, \xi, d) \right) \tag{4}$$

As the intra-application DSE only considers the required tiles of each type to execute a single considered application, it does not yet regard reliability, cost, and the embodied emissions, for which the total target architecture is needed. These missing objectives are included into the sustainability-aware inter-application DSE, described next.

4.2 Inter-Application DSE

The sustainability-aware inter-application DSE receives the optimized sets of constraint graphs H_a per application $a \in A$ and then explores sustainable target architectures and a selection of constraint graphs, which minimize the objectives of the mapping problem. This work assumes that the user specifies the location of the system at use, while the DSE explores the production location $\mu \in W$, incorporating the trade-off between cost and sustainability. Formally, the DSE optimizes a selection of constraint graphs $\eta' = \{\eta_{a_1}, ..., \eta_{a_{|A|}}\}$, and a *production location* μ such that the global goals of sustainability $Su(\eta', \xi, \mu)$ and reliability $Re(\xi)$ are maximized, and the mean deadline miss rate $DM(\eta', \xi)$ and the cost $Co(\xi, \mu)$ are minimized:

$$\underset{\eta', \xi, \mu}{\text{maximize}}(Su(\eta', \xi, \mu), Re(\xi), -DM(\eta', \xi), -Co(\xi, \mu)) \tag{5}$$

The architecture ξ is given implicitly by the union of resources allocated for each application, exploiting spatial isolation and composability [24]. In the following, each objective is discussed in more detail.

4.2.1 Sustainability

We maximize sustainability by minimizing the emissions $Su(\eta'_v, \xi, \mu)$ of the constraint graph vector η'_v on architecture ξ, estimated by the sum of the embodied emissions (scope 1 and scope 2) and the expected operational emissions over the lifetime of the MPSoC:

$$Su(\eta'_v, \xi, \mu) = -F_{\text{Scope1}}(\xi, \mu) - F_{\text{Scope2}}(\xi, \mu) - F_{\text{op}}(\eta'_v, \xi) \tag{6}$$

The *embodied scope 1* emissions F_{Scope1} can be estimated by the chip yield y, number of produced chips n_c, the number of wafers per chip $n_{\text{w/c}}$ for the target architecture ξ and the CO_2 emissions per wafer $E_{\text{CO}_2/\text{w}}$ at location μ (extended from [9]):

$$F_{\text{Scope1}}(\xi, \mu) = \frac{1}{y} \cdot \lceil n_c \cdot n_{\text{w/c}}(\xi) \rceil \cdot E_{\text{CO}_2/\text{w}}(\mu) \tag{7}$$

The chip yield y is the proportion of defect-free chips on a wafer, with defects being caused by contamination and other factors during manufacturing. The emissions per wafer $E_{\text{CO}_2/\text{w}}(\mu)$ depend on factors such as wafer size and semiconductor feature size. [17] reports a CO_2-equivalent GHG emission of 61 kg for a 300 mm wafer based on a 130 nm technology node, which can be reduced to about 13 kg of carbon equivalent by exhaust gas treatment, neutralizing or filtering out most of the NF_3 and CF_4 gases. [5] provides recent data on GHG emissions from wafer production, showing the gas quantities and compositions for different technology nodes (28 to 3 nm) and manufacturing processes. Emissions are greatly reduced by filtering out NF_3. The number of chips that can fit on a wafer depend on the wafer's diameter ζ and die size α_{die}:

$$n_{\text{w/c}}(\xi) = \frac{\pi \cdot \zeta^2}{4 \cdot \alpha_{\text{die}}(\xi)} - 0.58 \frac{\pi \cdot \zeta}{\sqrt{\alpha_{\text{die}}(\xi)}} \tag{8}$$

Since wafers are round and chips are rectangular, not all the wafer's area can be utilized, as estimated by the formula's second term. The die size $\alpha_{\text{die}}(\xi)$ is determined by the core area $\alpha_{\text{core}}(\xi)$, i.e., the area in which the SoC tiles on the architecture ξ are placed. By estimating the number of transistors for logic $n_{\text{tr,log}}$ and memory $n_{\text{tr,mem}}$ with respective transistor density χ_{log} and χ_{mem}, the core chip area amounts to:

$$\alpha_{\text{core}}(\xi) = g \cdot \left(\frac{n_{\text{tr,log}}(\xi)}{\chi_{\text{log}}} + \frac{n_{\text{tr,mem}}(\xi)}{\chi_{\text{mem}}}\right) \tag{9}$$

In this formula, g is the wiring factor, considering the additional space required for the connections between the transistors. However, the core area is surrounded by additional peripheral area containing on-chip components that do not correspond directly to the MPSoC architecture like, for instance, I/O pad cells. With the width of this overhead given as δ, the die area can then be estimated as:

$$\alpha_{\text{die}}(\xi) = (\lceil \sqrt{\alpha_{\text{core}}(\xi)} \rceil + \delta)^2 \tag{10}$$

The *embodied scope 2* emissions F_{Scope2} can be calculated as follows where eng_{w} denotes the needed energy to produce one wafer (extended from [9]):

$$F_{\text{Scope2}}(\xi, \mu) = \frac{1}{y} \cdot \lceil n_c \cdot n_{\text{w/c}}(\xi) \rceil \cdot eng_{\text{w}} \cdot \rho_\mu \tag{11}$$

The factor ρ_μ expresses the CO_2 intensity of the energy mix at location μ during production. [12] present the CO_2 efficiency of energy production in different regions. The energy required to produce a wafer depends on its size and the semiconductor's feature size. According

to [17], this includes energy for processing quartz into silicon, refining silicon into ingots, and further processing into wafers. Additional energy is needed for chemicals, semiconductor fabrication, and maintaining infrastructure. A separate study [10] provides further values across various semiconductor fabs. [5] presents the energy consumption for various semiconductor manufacturing processes (28 nm to 3 nm), assuming that the supporting infrastructure (e.g., cooling, vacuum) accounts for 40 % of total energy use. The *operational emissions* F_{op} can be computed as follows. Let $C \subseteq \mathbb{R} \times W$ be the set of tuples (β, λ) which describes that $\beta \cdot 100\,\%$ of the produced chips are expected to be operated in country λ. By considering both the ratio of chips $n_c \cdot \beta$ that are operated using a local energy mix ρ_λ and the energy consumption eng of the chip during its lifetime, see Eq. (3), the total operational emissions can be estimated as:

$$F_{\text{op}}(\eta'_v, \xi) = \sum_{(\beta,\lambda)\in C} (n_c \cdot \beta \cdot \rho_\lambda \cdot \sum_{\eta_a \in \eta'_v} eng(\eta_a, \xi)) \tag{12}$$

4.2.2 Reliability

The reliability $Re(\xi)$ of the architecture ξ models the mean time to failure (MTTF) of the system. We approximate the reliability as follows[1]:

$$Re(\xi) = \int_0^\infty \prod_{\psi\in\xi} \varsigma_\psi(t)dt = \int_0^\infty \prod_{\psi\in\xi} e^{-\lambda_\psi t}dt = \int_0^\infty e^{-t\sum_{\psi\in\xi}\lambda_\psi}dt = \frac{1}{\sum_{\psi\in\xi}\lambda_\psi} \tag{13}$$

In the above conservative model, it is assumed that the applications allocate all resources of the SoC system, and that the system is considered defective already once a single of those processing resources fails. ς_ψ denotes the reliability function of processing resource ψ and λ_ψ its failure rate. To determine the failure rate λ_ψ in Eq. (13), e.g. [16] can be used.

4.2.3 Performance

The performance $DM(\eta'_v, \xi)$ of the constraint graph vector η'_v on architecture ξ is estimated as follows. Given the set of periodic real-time applications where each application $a \in A$ has a priority p_a and a deadline θ_a, we want to minimize the average deadline misses $dm(\eta_a, \xi)$ of all applications that are mapped to the SoC:

$$DM(\eta'_v, \xi) = \frac{1}{|A|} \cdot \sum_{\eta_a\in\eta'_v} (1 - dm(\eta_a, \xi)) \cdot p_a \tag{14}$$

By dividing the sum of deadline misses of running applications by the total number of applications $|A|$, we reward systems implementing more applications than others with a higher performance $DM(\eta'_v, \xi)$.

4.2.4 Cost

The cost $Co(\xi)$ of an architecture ξ is modeled by the wafer cost κ_{w}, and the electricity cost κ_{el} per wafer:

$$Co(\xi) = \frac{\kappa_{\text{w}} + eng_{\text{w}} \cdot \kappa_{\text{el}}}{n_{\text{w/c}} \cdot y} \tag{15}$$

[1] Our reliability model (Eq. (13)) assumes independent failures of processing resources, equal load distribution, and all applications starting at the same time. While this simplification allows for efficient computation, it may overestimate system-level failure rates for architectures with redundancy. Future work can incorporate more complex reliability models to account for fault tolerance.

In this formula, $n_{\mathrm{w/c}}$ describes the number of chips per wafer (see Eq. (8)). The yield y of defect-free chips on a wafer is modeled using the negative binomial distribution, which accounts for defect clustering with the ϵ:

$$y = \left(1 + \frac{\alpha_{\mathrm{chip}} \cdot \chi_0}{\epsilon}\right)^{-\epsilon} \tag{16}$$

Here, α_{chip} is the critical chip area, χ_0 is the defect density. The defect density χ_0 depends on the production environment and structure size, with smaller structures more prone to defects. The cluster factor ϵ indicates defect distribution: higher values suggest evenly distributed defects (lower yield), while lower values indicate concentrated defects (higher yield). Typical values for ϵ range from 0.3 to 5 [15].

4.3 Optimization Time

The total optimization time of the proposed sustainability-aware DSE is composed of the optimization times of the intra-application DSEs, the inter-application DSE, and the negligible overhead t_{bind} of the binding algorithm. As the separate intra-application DSEs can be executed in parallel, their total optimization time is the maximum across the DSEs, which depends on the number of generations $\#\mathrm{gen}_a$ and the time per generation $t_{\mathrm{DSE_a,ge}}$. The optimization time for the inter-application sustainability-aware DSE is given by the number of generations $\#\mathrm{gen}_{\mathrm{sus}}$ and the time per generation $t_{\mathrm{sus,ge}}$:

$$t_{\mathrm{total}} = \#\mathrm{gen}_{\mathrm{sus}} \cdot t_{\mathrm{sus,ge}} + t_{\mathrm{bind}} + \max_{a \in A} \left(\#\mathrm{gen}_a \cdot t_{\mathrm{DSE_a,ge}}\right) \tag{17}$$

In our experiments, we performed the inter-application DSE for 1,000 generations, with $t_{\mathrm{sus,ge}} = 20\,\mathrm{ms}$. For the intra-application DSE, we also performed 1,000 generations, where the time per generation depends on the number of new individuals created in this generation and the time for evaluating an individual. Assessing the execution time of an individual requires simulating the execution of the application under the individual mapping on the platform, which takes up to half a second. For input-dependent applications, a separate execution is needed for every training data, leading up to even higher execution times. The optimization time is thus typically dominated by the intra-application phase, while the inter-application DSE is highly efficient and only takes around 2 seconds. Hence, the proposed approach is highly scalable and can be used to design large and complex embedded systems.

5 Evaluation

The proposed sustainable-aware DSE is highly customizable and can be fine-tuned by the user. This customization allows a flexible approach across different production scenarios, but requires accurate settings for a reliable result. Since these parameters are often classified and not openly available, we resort in this work to rather limited publicly available data, e.g., for modeling the sustainability objectives, we assumed a lifetime of 5 years, used the emission data from [12, 17, 12, 10, 5], and extracted reliability data from [16]. The lack of accurate data does, however, not invalidate our results, since although different settings may alter the design space, the superiority of our approach in exploring the entire design space remains consistent across various parameter configurations. In the following evaluation, the ray-tracing, Mjpeg, and stitching application from [24], combined with the auto consumer, auto industry, office automation, and telecom applications from [8] serve

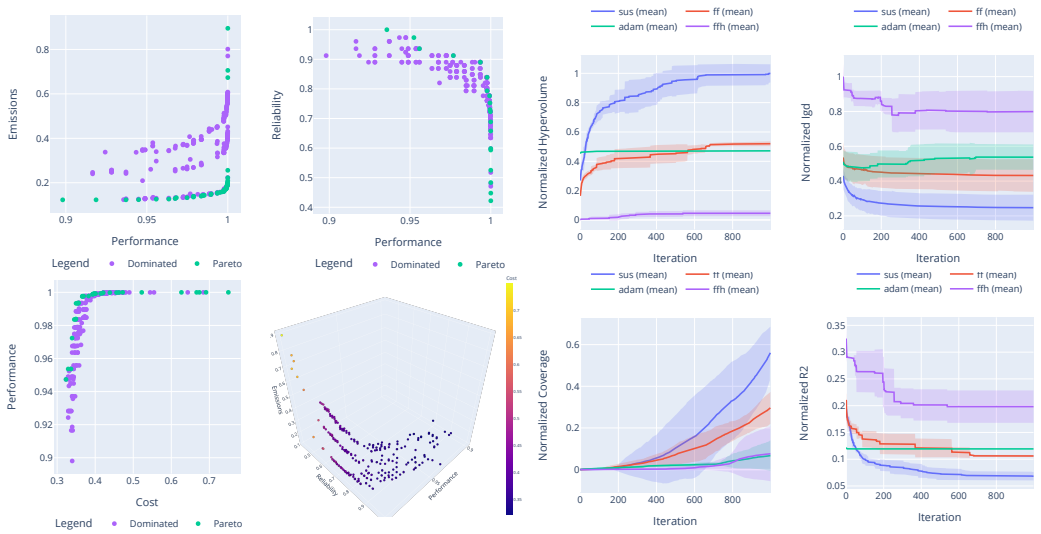


Figure 3 Different objective trade-offs of the obtained approximation of the Pareto front by the proposed sustainability-aware DSE (left) and evolution of the normalized hypervolume [4], IGD [30], R2 [26], and coverage [31] metric for FirstFit (ff), AdaMD+ (adam), Composition (ffh), and the proposed sustainability-aware DSE (sus) (right). Shown are the average results across 5 runs (solid line) with the shaded area representing the ±1 standard deviation from the mean.

as the soft real-time applications A that shall be executed on a SoC target architecture. All DSEs are implemented in Opt4J [18]. Fig. 3 shows on the left the embedded system solutions explored by the proposed multi-objective sustainability-aware DSE concerning the objectives of emissions, reliability, performance and cost. To facilitate comparability, the objectives are normalized in the following. The comparison of emissions and performance shows a clear trade-off, where higher performance correlates with increased emissions. In the mid-performance range, significant performance improvements can be achieved with only small increases in emissions. This finding suggests that it's possible to design systems with high performance without necessarily incurring large environmental costs. However, the graph shows a sharp increase in emissions at the highest performance levels. This steep rise indicates that aiming for maximum performance leads to disproportionately higher emissions. The data suggests that by slightly reducing performance targets from their maximum, designers can potentially achieve large reductions in emissions, leading to more environmentally friendly systems without major performance losses. The normalized reliability versus performance plot demonstrates an inverse relationship. As performance increases, reliability tends to decrease, with the Pareto front showing a steep drop in reliability at the highest performance levels. This suggests a critical trade-off point where marginal performance gains come at

Table 1 Normalized key metrics upon convergence.

Metric	R2 (↓)	Hypervolume (↑)	IGD (↓)	Coverage (↑)
Composition	0.20	0.05	0.80	0.08
AdaMD+	0.12	0.47	0.53	0.07
FirstFit	0.11	0.52	0.43	0.30
Proposed	**0.07**	**1.0**	**0.25**	**0.56**

a substantial cost to reliability. Finally, the performance versus cost plot exhibits a clear positive correlation, showing how higher performance invariably comes at a greater cost. At the highest performance levels, the cost increases sharply. These visualizations indicate the complexity of the design space and the importance of a DSE approach in identifying non-obvious trade-offs that might be overlooked by simpler methods. To further justify the need for a DSE incorporating sustainability aspects, We compare the proposed sustainability-aware DSE against various baselines regarding hypervolume, inverted generational distance (IGD), coverage, and r2 indicator [31]. This includes *FirstFit*, an extension of the multi-objective DSE from [24] with reliability and cost objectives, *AdaMD+*, an extension of the fast multi-application mapping optimization methodology by [20], and *Composition*, which uses *AdaMD* to determine the architecture as a composition of the mappings per application. This evaluation is based on the emission data from [12, 17, 12, 10, 5]. Acc. to Fig. 3, our proposed approach demonstrates superior performance across all metrics. The hypervolume metric rapidly converges to 1.0, while the best baseline, FirstFit, plateaus at 0.5. The IGD plot also shows that our solutions are consistently closer to the true Pareto front. Our method further excels in the coverage metric, finding nearly twice as many non-dominated solutions as FirstFit. Finally, the R2 indicator validates our method's superior convergence and diversity. Table 1 lists the final R2, hypervolume, IGD, and coverage values upon convergence.

6 Summary

As the demand for computing continues to grow, the environmental impact of this development is often neglected, despite the fact that data centers already contribute massively to global emissions. However, designing an environmentally friendly embedded SoC system without compromising performance, reliability, or cost is quite complex, as all objectives must be carefully balanced. As a solution, this work proposes a multi-objective, sustainability-aware two-phase DSE to find Pareto-optimal tile-based MPSoC architectures for a set of given time-critical applications. In an intra-application phase, the DSE first determines optimal mappings for each individual application in the form of constraint graphs, and then explores the combination of these mappings in an inter-application phase. Here, the DSE is sustainability-aware by considering the embodied *scope 1* and *scope 2* emissions of the architecture during chip fabrication and its operation as objective. In addition, the DSE also considers the reliability, manufacturing cost, i.e., the die area, and the performance on the set of applications. In a thorough evaluation against various baselines from the literature, the proposed work dominates over four selected metrics significantly. Particularly, it achieves nearly double the hypervolume than the best baseline, indicating that it explores a wider part of the design space, determines solutions closer to the true Pareto front, and explores a more diverse set of non-dominated solutions.

References

1 Athena Abdi and Hamid R. Zarandi. HYSTERY: A Hybrid Scheduling and Mapping Approach to Optimize Temperature, Energy Consumption and Lifetime Reliability of Heterogeneous Multiprocessor Systems. *J. Supercomput.*, 74(5):2213–2238, 2018. doi:10.1007/s11227-018-2248-2.

2 Benny Akesson, Anca Mariana Molnos, Andreas Hansson, Jude Angelo Ambrose, and Kees Goossens. Composability and Predictability for Independent Application Development, verification, and execution. In Michael Hübner and Jürgen Becker, editors, *Multiprocessor System-on-Chip - Hardware Design and Tool Integration*, pages 25–56. Springer, 2011. doi:10.1007/978-1-4419-6460-1_2.

3 Nidhi Anantharajaiah, Tamim Asfour, Michael Bader, Lars Bauer, Jürgen Becker, Simon Bischof, Marcel Brand, Hans-Joachim Bungartz, Christian Eichler, Khalil Esper, Joachim Falk, Nael Fasfous, Felix Freiling, Andreas Fried, Michael Gerndt, Michael Glaß, Jeferson Gonzalez, Frank Hannig, Christian Heidorn, Jörg Henkel, Andreas Herkersdorf, Benedict Herzog, Jophin John, Timo Hönig, Felix Hundhausen, Heba Khdr, Tobias Langer, Oliver Lenke, Fabian Lesniak, Alexander Lindermayr, Alexandra Listl, Sebastian Maier, Nicole Megow, Marcel Mettler, Daniel Müller-Gritschneder, Hassan Nassar, Fabian Paus, Alexander Pöppl, Behnaz Pourmohseni, Jonas Rabenstein, Phillip Raffeck, Martin Rapp, Santiago Narváez Rivas, Mark Sagi, Franziska Schirrmacher, Ulf Schlichtmann, Florian Schmaus, Wolfgang Schröder-Preikschat, Tobias Schwarzer, Mohammed Bakr Sikal, Bertrand Simon, Gregor Snelting, Jan Spieck, Akshay Srivatsa, Walter Stechele, Jürgen Teich, Isaías A. Comprés Ureña, Ingrid Verbauwhede, Dominik Walter, Thomas Wild, Stefan Wildermann, Mario Wille, Michael Witterauf, and Li Zhang. *Invasive Computing*. FAU University Press, 2022. doi:10.25593/978-3-96147-571-1.

4 Anne Auger, Johannes Bader, Dimo Brockhoff, and Eckart Zitzler. Hypervolume-based multiobjective optimization: Theoretical foundations and practical implications. *Theo. Comp. Sci.*, 425:75–103, 2012. doi:10.1016/J.TCS.2011.03.012.

5 M Garcia Bardon, P Wuytens, L-Å Ragnarsson, G Mirabelli, D Jang, G Willems, A Mallik, A Spessot, J Ryckaert, and B Parvais. DTCO including sustainability: Power-performance-area-cost-environmental score (PPACE) analysis for logic technologies. In *2020 IEDM*, pages 41–4. IEEE, 2020.

6 Tilera Corporation. Tile Processor Architecture Overview for the Tile-Gx Series. https://cdn.manesht.ir/17871___210769647-UG130-ArchOverview-TILE-Gx.pdf, 2012. [Online; accessed 04-July-2023].

7 Benoît Dupont de Dinechin, Renaud Ayrignac, Pierre-Edouard Beaucamps, Patrice Couvert, Benoit Ganne, Pierre Guironnet de Massas, François Jacquet, Samuel Jones, Nicolas Morey Chaisemartin, Frédéric Riss, and Thierry Strudel. A Clustered Manycore Processor Architecture for Embedded and Accelerated Applications. In *HPEC 2013, Waltham, MA, USA, September 10-12, 2013*, pages 1–6. IEEE, 2013. doi:10.1109/HPEC.2013.6670342.

8 Robert P. Dick. Embedded System Synthesis Benchmarks Suite (E3S), 2002. Accessed September 11, 2024. URL: https://ziyang.eecs.umich.edu/~dickrp/e3s/.

9 Lieven Eeckhout. Kaya for Computer Architects: Toward Sustainable Computer Systems. *IEEE Micro*, 43(1):9–18, 2023. doi:10.1109/MM.2022.3218034.

10 Robert Falkner. The Paris Agreement and the new logic of international climate politics. *International Affairs*, 92(5):1107–1125, 2016.

11 Charlotte Freitag, Mike Berners-Lee, Kelly Widdicks, Bran Knowles, Gordon S. Blair, and Adrian Friday. The real climate and transformative impact of ICT: A critique of estimates, trends, and regulations. *Patterns*, 3(8):100576, 2022. doi:10.1016/J.PATTER.2022.100576.

12 Udit Gupta, Young Geun Kim, Sylvia Lee, Jordan Tse, Hsien-Hsin S Lee, Gu-Yeon Wei, David Brooks, and Carole-Jean Wu. Chasing carbon: The elusive environmental footprint of computing. In *HPCA*, pages 854–867. IEEE, 2021. doi:10.1109/HPCA51647.2021.00076.

13 Jan Heisswolf, Ralf König, and Jürgen Becker. A Scalable NoC Router Design Providing QoS Support Using Weighted round robin scheduling. In *ISPA, Leganes, Madrid, Spain, July 10-13, 2012*, pages 625–632. IEEE Computer Society, 2012. doi:10.1109/ISPA.2012.93.

14 Jason Howard, Saurabh Dighe, Sriram R. Vangal, Gregory Ruhl, Nitin Borkar, Shailendra Jain, Vasantha Erraguntla, Michael Konow, Michael Riepen, Matthias Gries, Guido Droege, Tor Lund-Larsen, Sebastian Steibl, Shekhar Borkar, Vivek K. De, and Rob F. Van der Wijngaart. A 48-Core IA-32 Processor in 45 nm CMOS Using On-Die Message-Passing and DVFS for performance and power scaling. *IEEE J. Solid State Circuits*, 46(1):173–183, 2011. doi:10.1109/JSSC.2010.2079450.

15 Jung Yoon Hwang. *Spatial stochastic processes for yield and reliability management with applications to nano electronics*. Texas A&M University, 2004.

16 Texas Instruments. MTBF and FIT Rate Estimator, 2024. URL: https://www.ti.com/quality/docs/estimator.tsp.

17 Nikhil Krishnan, Sarah Boyd, Ajay Somani, Sebastien Raoux, Daniel Clark, and David Dornfeld. A hybrid life cycle inventory of nano-scale semiconductor manufacturing. *Environmental science & technology*, 42(8):3069–3075, 2008.

18 Martin Lukasiewycz, Michael Glaß, Felix Reimann, and Jürgen Teich. Opt4J: A Modular Framework for Meta-Heuristic Optimization. In *GECCO, Dublin, Ireland, July 12-16*, pages 1723–1730. ACM, 2011. doi:10.1145/2001576.2001808.

19 Behnaz Pourmohseni, Michael Glaß, Jörg Henkel, Heba Khdr, Martin Rapp, Valentina Richthammer, Tobias Schwarzer, Fedor Smirnov, Jan Spieck, Jürgen Teich, Andreas Weichslgartner, and Stefan Wildermann. Hybrid Application Mapping for Composable Many-Core Systems: Overview and Future Perspective. *Journal of Low Power Electronics and Applications*, 10:1–37, 2020. doi:10.3390/jlpea10040038.

20 Basireddy Karunakar Reddy, Amit Kumar Singh, Bashir M. Al-Hashimi, and Geoff V. Merrett. AdaMD: Adaptive Mapping and DVFS for Energy-Efficient Heterogeneous multicores. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 39(10):2206–2217, 2020. doi:10.1109/TCAD.2019.2935065.

21 Tobias Schwarzer, Andreas Weichslgartner, Michael Glaß, Stefan Wildermann, Peter Brand, and Jürgen Teich. Symmetry-Eliminating Design Space Exploration for Hybrid Application mapping on many-core architectures. *TCAD*, 37(2):297–310, 2018. doi:10.1109/TCAD.2017.2695894.

22 Jan Spieck, Stefan Wildermann, and Jürgen Teich. On Transferring Application Mapping Knowledge Between Differing MPSoC architectures. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 41(11):4289–4300, 2022. doi:10.1109/TCAD.2022.3197527.

23 Jan Spieck, Stefan Wildermann, and Jürgen Teich. A Learning-based Methodology for Scenario-aware Mapping of Soft Real-time applications onto heterogeneous mpsocs. *ACM Trans. Design Autom. Electr. Syst.*, 28(1):4:1–4:40, 2023. doi:10.1145/3529230.

24 Jan Spieck, Stefan Wildermann, and Jürgen Teich. A Scenario-Based DVFS-Aware Hybrid Application Mapping Methodology for MPSoCs. *ACM Trans. Des. Autom. Electron. Syst.*, 29(4), June 2024. doi:10.1145/3660633.

25 Chetan Choppali Sudarshan, Nikhil Matkar, Sarma B. K. Vrudhula, Sachin S. Sapatnekar, and Vidya A. Chhabria. ECO-CHIP: Estimation of Carbon Footprint of Chiplet-based Architectures for sustainable VLSI. In *HPCA, Edinburgh, United Kingdom, March 2-6, 2024*, pages 671–685. IEEE, 2024. doi:10.1109/HPCA57654.2024.00058.

26 Heike Trautmann, Tobias Wagner, and Dimo Brockhoff. R2-EMOA: Focused Multiobjective Search Using R2-Indicator-Based selection. In Giuseppe Nicosia and Panos M. Pardalos, editors, *LION 7, Catania, Italy, January 7-11, 2013, Revised Selected Papers*, volume 7997, pages 70–74. Springer, 2013. doi:10.1007/978-3-642-44973-4_8.

27 Peter van Stralen and Andy D. Pimentel. Scenario-Based Design Space Exploration of MPSoCs. In *ICCD 2010, 3-6 October 2010, Amsterdam, The Netherlands, Proceedings*, pages 305–312, 2010. doi:10.1109/ICCD.2010.5647727.

28 Dominik Walter, Marcel Brand, Christian Heidorn, Michael Witterauf, Frank Hannig, and Jürgen Teich. ALPACA: an Accelerator Chip for Nested Loop Programs. In *IEEE International Symposium on Circuits and Systems, ISCAS 2024, Singapore, May 19-22, 2024*, pages 1–5. IEEE, 2024. doi:10.1109/ISCAS58744.2024.10558549.

29 World Resources Institute, C40 Cities Climate Leadership Group, and ICLEI Local Governments for Sustainability. *Global Protocol for Community-Scale Greenhouse Gas Emission Inventories*. World Resources Institute, Washington, DC, 2014. [Online; accessed 2024-09-16].

30 Gary G. Yen and Zhenan He. Performance Metric Ensemble for Multiobjective Evolutionary Algorithms. *IEEE Trans. Evol. Comput.*, 18(1):131–144, 2014. doi:10.1109/TEVC.2013.2240687.

31 Eckart Zitzler and Lothar Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach. *IEEE Trans. Evol. Comput.*, 3(4):257–271, 1999. doi:10.1109/4235.797969.

H-MBR: Hypervisor-Level Memory Bandwidth Reservation for Mixed Criticality Systems

Afonso Oliveira
Centro ALGORITMI / LASI, Universidade do Minho, Portugal

Diogo Costa
Centro ALGORITMI / LASI, Universidade do Minho, Portugal

Gonçalo Moreira
Centro ALGORITMI / LASI, Universidade do Minho, Portugal

José Martins
Centro ALGORITMI / LASI, Universidade do Minho, Portugal

Sandro Pinto
Centro ALGORITMI / LASI, Universidade do Minho, Portugal

Abstract

Recent advancements in fields such as automotive and aerospace have driven a growing demand for robust computational resources. Applications that were once designed for basic Microcontroller Units (MCUs) are now deployed on highly heterogeneous System-on-Chip (SoC) platforms. While these platforms deliver the necessary computational performance, they also present challenges related to resource sharing and predictability. These challenges are particularly pronounced when consolidating safety-critical and non-safety-critical systems, the so-called Mixed-Criticality Systems (MCS) to adhere to strict Size, Weight, Power, and Cost (SWaP-C) requirements. MCS consolidation on shared platforms requires stringent spatial and temporal isolation to comply with functional safety standards (e.g., ISO 26262). Virtualization, mainly leveraged by hypervisors, is a key technology that ensures spatial isolation across multiple OSes and applications; however ensuring temporal isolation remains challenging due to contention on shared resources, such as main memory, caches, and system buses, which impacts real-time performance and predictability. To mitigate this problem, several strategies (e.g., cache coloring and memory bandwidth reservation) have been proposed. Although cache coloring is typically implemented on state-of-the-art hypervisors, memory bandwidth reservation approaches are commonly implemented at the Linux kernel level or rely on dedicated hardware and typically do not consider the concept of Virtual Machines that can run different OSes. To fill the gap between current memory bandwidth reservation solutions and the deployment of MCSs that operate on a hypervisor, this work introduces *H-MBR*, an open-source VM-centric memory bandwidth reservation mechanism. *H-MBR* features (i) VM-centric bandwidth reservation, (ii) OS and platform agnosticism, and (iii) reduced overhead. Empirical results evidenced no overhead on non-regulated workloads, and negligible overhead (<1%) for regulated workloads for regulation periods of 2 µs or higher.

2012 ACM Subject Classification Computer systems organization → Real-time system specification; Computer systems organization → Embedded software

Keywords and phrases Virtualization, Multi-core Interference, Mixed-Criticality Systems, Arm, Memory Bandwidth Reservation

Digital Object Identifier 10.4230/OASIcs.NG-RES.2025.4

Supplementary Material *Software (Source Code)*: `https://gitlab.com/ESRGv3/h-mbr`

Funding *Diogo Costa*: Supported by FCT grant 2022.13378.BD.
José Martins: Supported by FCT grant SFRH/BD/138660/2018.

1 Introduction

Rapid advancements in industries such as automotive, aerospace, and industrial automation have led to increasingly demanding applications, driving a significant need for higher computational power [8, 5]. These applications now require systems capable of handling a diverse range of computationally intensive tasks, such as autonomous driving, flight control, and complex industrial automation workflows [16]. To address this computational needs, the once simple microcontroller units (MCUs) evolved to sophisticated heterogeneous architectures. Modern designs integrate multiple Central Processing Units (CPUs) alongside specialized accelerators, such as Graphics Processing Units (GPUs)[5], Field-Programmable Gate Arrays (FPGAs)[16], and novel AI accelerators like Tensor Processing Units (TPUs) and Neural Processing Units (NPUs)[21, 14]. This shift enables efficient processing of complex workloads but simultaneously introduces challenges related to resource sharing and system predictability, especially as systems grow in complexity and criticality[30]. In addition to computational demands, embedded systems increasingly face Size, Weight, Power, and Cost (SWAP-C) constraints, which are driving a trend towards system consolidation[8, 13], giving rise to the so called Mixed-Criticality Systems (MCS). However, integrating MCS on shared platforms faces a significant challenge: stringent requirements for both spatial and temporal isolation [40, 16, 1, 22, 12]. These systems must adhere to certification standards (e.g., ISO 26262 in automotive, CENELEC for railway systems, and ECSS for space applications) which place strict demands on system safety, reliability, and predictability [25, 6, 11, 34, 20].

Virtualization has become essential in the consolidation of MCS, with hypervisors playing a pivotal role. Hypervisors, particularly static partitioning hypervisors, must be minimal yet sufficiently robust to ensure reliable isolation while meeting the real-time (RT) performance requirements of modern applications [23, 11, 29, 26]. Despite advancements in spatial isolation through various partiotioning techniques, temporal isolation remains an active area of research due to contention on shared resources [39, 40, 25]. These shared resources, including (i) main memory, (ii) caches, and (iii) the system bus, can introduce substantial unpredictability in response times if left unregulated [6, 30, 12]. Such variability in access latency poses significant risks to RT, safety-critical applications, where predictable performance is essential to meet strict timing constraints [17, 28, 20]. To address shared resource contention, considerable efforts have been made in both academia and industry. Techniques such as (i) cache coloring and (ii) MBR have emerged as widely recognized approaches to regulate and reduce contention induced by memory access [40, 39, 22]. However, unlike cache coloring, most MBR approaches are implemented at the OS-level, typically on the Linux kernel [12, 25], narrowing their flexibility and applicability, underscoring the need for a more versatile solution, one that can provide robust temporal isolation across diverse applications and platforms without sacrificing configurability or performance [7, 19, 6].

In this paper we present H-MBR, an open-source VM-centric MBR mechanism implemented in Bao hypervisor. Its primary contributions over traditional MBR methods stem from an innovative design: (i) VM-Centric bandwidth allocation, (ii) OS and platform agnosticism, and (iii) reduced overhead. Furthermore, H-MBR supports unbalanced distribution of bandwidth across a VM's vCPUs. As H-MBR's is implemented at the hypervisor level it enables the memory bandwidth regulation of VMs with different operating systems, (e.g., Linux, FreeRTOS and Zephyr). Additionally, H-MBR is designed to be platform-agnostic, allowing it to be ported across various architectures such as ARMv8-A, ARMv8-R, and RISC-V . Finally, as Bao hypervisor stands as the hypervisor with lowest interrupt latency among the state-of-the-art static partitioning hypervisors [24], this mechanism takes advantage of

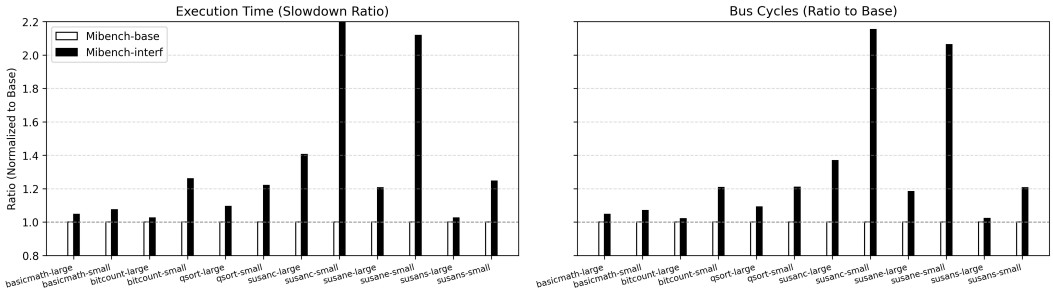


Figure 1 Mibench Execution Time and Bus Cycles (Ratio).

the fast interrupt handling, which minimizes the overhead of MBR. Empirical results show that *H-MBR* introduces no overhead for critical workloads, maintaining their performance and ensuring predictable behavior. For non-critical workloads, the mechanism introduces an overhead below 1% for regulation periods equal or higher than 2 µs. It also fully eliminates critical workload performance degradation if strict budgets are enforced on non-critical workloads, demonstrating its reliability in MCSs.

2 Interference and Interference Mitigation

In multicore embedded systems, contention generated in shared resources is a significant challenge that impacts performance, predictability, and temporal isolation, particularly in RT and MCS [7, 40, 6]. Contention arises when multiple cores simultaneously access shared components [23, 34], (i) such as main memory [42, 33, 41, 20, 40], (ii) last-level caches [23, 22], (iii) system buses [33], and (iv) additional subsystems like the Generic Interrupt Controller [9, 10]. These shared resources can lead to delays as cores compete for access, resulting in increased response times and unpredictable behavior. This unpredictability is especially problematic in systems with strict timing requirements, where such delays affect the system's temporal isolation, threatening reliability and safety [12, 22, 11]. As shown in Figure 1, this interference can arise up to 2.3x on memory-intensive benchmarks like in Mibench's susanc-small. As system complexity grows, understanding and controlling memory utilization becomes essential to mitigate interference effects and ensure critical tasks' predictability. To mitigate memory-induced contention and ensure temporal isolation of tasks, several techniques have been proposed.

Cache Partitioning. Cache partitioning divides LLC into distinct regions, or "partitions", which are assigned to specific workloads to control cache access and reduce contention. Two primary approaches to cache partitioning are (i) cache locking and (ii) cache coloring. Cache locking relies on hardware assistance to restrict eviction from designated cache lines, securing specific portions of the cache for high-priority tasks. Cache coloring, on the other hand, uses the overlap between virtual page numbers and cache indices to partition cache sets without needing specialized hardware. Hybrid techniques, such as Colored Lockdown [22], combine coloring and locking, while other approaches propose dynamic re-coloring schemes to adapt to varying workloads [39, 38, 31]. Cache coloring has been successfully implemented in hypervisors like Bao [23], Jailhouse [20], XVisor [25], Xen [32] and KVM [35], effectively enhancing predictability in multicore systems with real-time requirements. However, the efficiency of cache coloring is workload-dependent: for high-demand cores, it significantly reduces cache interference and improves performance, but with lighter or variable loads, it

Table 1 Gap analysis.

Paper	Dynamic	Static	Implementation Level	CPU-centric	VM-centric
A. Zuepke el al. [45]	●	○	Firmware	●	○
H.Yun et al.[41]	●	○	OS(Linux Kernel)	●	○
H.Yun el al. [43]	●	○	OS(Linux Kernel)	●	○
H.Yun et al.[40]	●	○	OS(Linux Kernel)	●	○
P. Sohal et al.[36]	●	○	Hardware	●	○
A. Agrawal et al.[2]	●	○	OS(Linux Kernel)	●	○
E. Seals [33]	●	○	OS(Linux Kernel)	●	○
D. Hoornaert et al. [18]	●	○	Hypervisor and Hardware	●	○
M. Pagani et al. [27]	●	○	Hardware	●	○
M. Xu et al.[37]	●	○	Hypervisor	●	○
P. Modica et al.[25]	○	●	Hypervisor	●	○
E. Gomes et al.[15]	●	○	Hypervisor	●	○
G. Brilli et al.[4]	●	○	Hypervisor	●	○
H-MBR	○	●	Hypervisor	○	●

may underutilize cache space, potentially impacting overall system efficiency. Additionally, cache coloring is hardware-specific, requiring an understanding of cache structure details such as size, associativity, and mapping policies, which can limit its portability across platforms.

Memory Bandwidth Reservation (MBR). MBR is a critical approach to reserving memory bandwidth, designed to prevent hardware contention on memory accesses, which is vital for achieving predictable performance in MCS [12, 40]. When memory is heavily utilized, increased memory contention leads to bus delays, subsequently extending execution times. MBR works by reserving memory bandwidth quotas per core or VM, either statically or dynamically, effectively reducing contention on shared resources and improving overall performance [7, 25]. Most MBR implementations operate at the OS level [41, 22, 40, 6, 34, 12, 44, 3] particularly within the Linux Kernel, where they manage bandwidth based on task requirements to ensure that critical tasks receive adequate access without overloading the memory controller [6, 34]. However, this OS-level integration limits MBR's portability to other OSs, such as RTOSs like Zephyr and FreeRTOS, which are especially prevalent in real-time applications [17, 29]. On the other hand, some MBR implementations use FPGA-dedicated accelerators [44] to monitor and control memory bandwidth, which has the significant downside of being platform-specific. Additionally, some approaches implement MBR at the hypervisor level [25, 37, 15, 4]; however, existing methods are applied at the vCPU level and are integrated into hypervisors with slower interrupt handling compared to Bao [24]. This slower response introduces significant overhead, particularly for time-sensitive applications, where Bao's efficient, low-latency interrupt handling offers a clear advantage. Additionally, some of this approaches are tied to specific architecture, reducing it's portability. Table 1 compares various MBR techniques, highlighting that most Linux OS-based implementations are dynamically managed yet tied to the Linux kernel, restricting their flexibility and scalability across diverse embedded environments.

Gap Analysis. MemGuard [41] presents a dynamic allocation in Linux kernel while H.Yun et al. [43] presents an extension to previous implementation. Palloc[40] implements a bank-aware memory allocation strategy. A. Zuepke et al. [45] adopt a distinct methodology

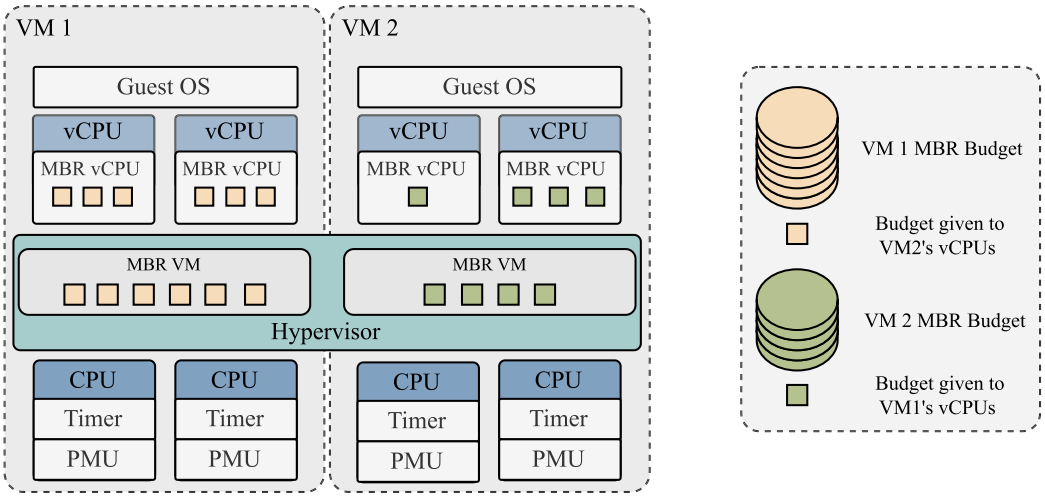


Figure 2 System Overview.

by utilizing an MCU to control a larger APU. A. Agrawal et al. [2], and E. Seals [33] all implement dynamic allocation within the Linux kernel. Hardware-level implementations come from P. Sohal et al.[36] and M. Pagani et al.[27], both using dynamic allocation. At the hypervisor level, E. Gomes et al. [15] and G. Brilli et al. [4] leverage MemGuard on the hypervisor level, focusing on CPU-centric approaches. Moreover, M. Xu et al.[37] implement dynamic allocation, while P. Modica et al.[25] opt for static allocation. D. Hoornaert et al. [18] uniquely combine hypervisor and hardware approaches with dynamic allocation. While all these solutions focus on CPU-centric approaches, *H-MBR* introduces a VM-centric approach that aligns with MCSs' perspective by enabling bandwidth allocation based on VMs than individual cores, implementing static allocation at the hypervisor level.

3 The mechanism

H-MBR Interface

H-MBR focuses on a VM-centric perspective, where each VM requires two key MBR parameters: (i) *budget* and (ii) *period* configured at compile time. The MBR configuration operates on a per-VM basis, enabling customized budgets and regulation periods for individual VMs. Additionally, the VM budget distribution across CPUs can be handled in two distinct ways: through automatic balancing or via non-balanced distribution, where specific allocations can be manually set. This flexibility allows for uneven distribution of the VM budget across its vCPUs, as demonstrated by the green-colored VM in Figure 2.

H-MBR Run-Time

The MBR mechanism follows a Memguard [41] based approach on how to track this metrics in real-time, thereby the mechanism relies on two key peripherals: (i) PMU and (ii) generic timer, respectively. Both of this peripherals exist physically inside each CPU, as depicted in figures 3 and 4. The hypervisor-level approach leverages this physical peripherals common to most architectures and assigns them directly to each vCPU, which leads to CPU monitoring not impacting other CPUs performance and therefore ensuring spatial and temporal isolation.

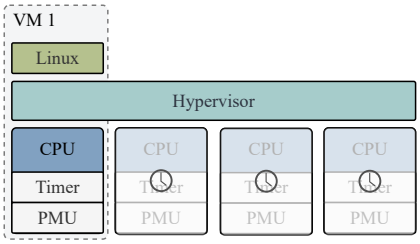


Figure 3 Standalone setup with single Linux VM.

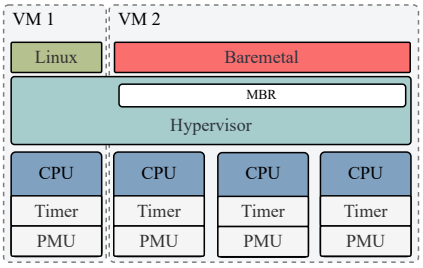


Figure 4 Co-existing setup with two VMs: (i) a Linux VM and (ii) a baremetal VM.

Timer. The timer is configured to overflow after a defined *Period*, setting the interval at which the memory bandwidth reserved for each core is reset. This periodic reset ensures that all cores start each *period* with a reset *budget*. Additionally, the timer re-activates any cores that were previously idled due to over-budget usage, allowing them to resume operation at the beginning of each new period. The overhead introduced by the timer interrupt can be quantified as follows: $\text{Overhead}_{\text{timer}} = \frac{D_{\text{timer}}}{\text{Period}_{\text{timer}}}$, where D_{timer} corresponds to the total execution time of the timer interrupt (i.e., the sum of interrupt injection latency with the interrupt callback execution time).

PMU. The PMU tracks memory access activities by counting *bus access* and triggers an overflow interrupt if a core exceeds its reserved *Budget* within the *Period*. When this overflow occurs, the PMU signals the MBR mechanism to temporarily idle the over-budget core. This action prevents excessive memory contention, helping to maintain consistent response times for critical tasks on other cores.

Furthermore, the MBR configuration which operates on a per-VM basis, enables customized budgets and regulation periods for individual VMs. The VM budget distribution across CPUs can be handled in two distinct ways: through automatic balancing or via non-balanced distribution, where specific allocations can be manually set.

4 Evaluation

This section provides a detailed explanation of the methodology used to obtain the results, including descriptions of the guest OSs, configurations, and setups on the target hardware. Each component was carefully chosen to analyze memory interference under varying conditions.

Algorithm 1 Initial Assignment of MBR Parameters.

```
for each VM in VMs do
  VM.budget ← budget_vm
  VM.period ← period_vm
  for each vCPU in VM.vCPUs do
    vCPU.budget ← VM.has_custom_dist
          ?  (VM.budget × vCPU_percentage[vCPU])
          :  (VM.budget / VM.num_vCPUs)
  end for
end for
```

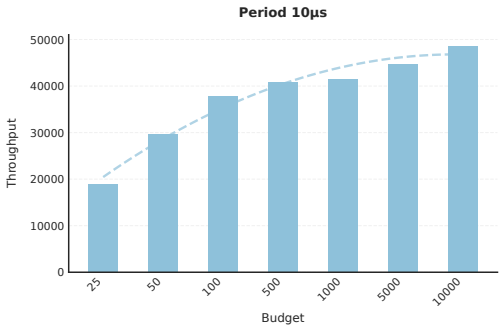


Figure 5 *interf+mbr* NC throughput budget variation.

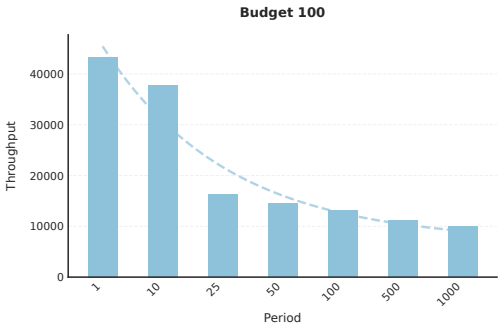


Figure 6 *interf+mbr* NC throughput period variation.

Target Platform & Measurement Tools. We conducted our experiments assessment on a Zynq UltraScale+ ZCU104 board (ARMv8-A-based architecture), which features a quad-core Arm Cortex-A53 processor. Each CPU has a private cache (data and instruction, 32KiB each), and the unified shared L2 cache (1MiB). The Cortex-A53 processor also features a ARM PMUv3, which has been leveraged to profile benchmark execution and gather key microarchitectural events. The selected events include bus cycles and execution cycles, providing insight into memory and system bus usage. Additionally, we used the `perf` tool on the Linux OS as an interface to the CPU's PMU.

VM Workloads. We deployed two distinct guest environments for benchmarking: (i) a Linux-based VM, which will be denominated as Critical VM (C) and (ii) a baremetal VM, which will be denoted as Non-Critical VM (NC). For the critical VM, we deployed a Linux-based guest to enable the deployment of MiBench automotive suite [17], a widely-used benchmark for automotive embedded systems – to simulate real-world automotive workloads. For the NC VM, we deployed a baremetal guest that continuously writes to a buffer sized to match the LLC capacity, creating intentional memory contention to thoroughly assess interference effects on the memory hierarchy. We focused exclusively on write operations, rather than reads or a combination of both, since write operations stresses even more the memory shared hardware resources, when compared to read operations.

Setups and Configurations. We evaluated four distinct setups: (i) *solo*, (ii) *interf*, and (iii)*interf+mbr*. The *solo* configuration involves running Linux alone, whereas *interf* includes both Linux and Baremetal guests to introduce memory interference. The suffix *mbr* is used to identify setups with memory bandwidth reservation. As for hardware resources, we followed the CPU assignment identified in Figure 4: one CPU is allocated to Linux in all configurations, and three CPUs are assigned to the baremetal VM, when included.

4.1 Impact of MBR on Baremetal Throughput and Overhead

To assess the effect of MBR configurations on the non-critical baremetal VM, we measured the number of cache line writes achievable under different budget and period settings. This section will focus on the memory-intensive *susan-c* benchmark from the MiBench suite(the results of other benchmarks can be found in Appendix A) on the critical Linux-based VM. The *susan-c* benchmark was specifically chosen for its high memory contention characteristics, making it an ideal workload for evaluating the impact of MBR adjustments on memory access behavior in the non-critical VM.

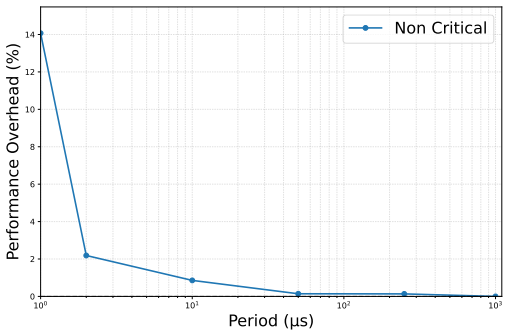


Figure 7 Performance overhead on NC.

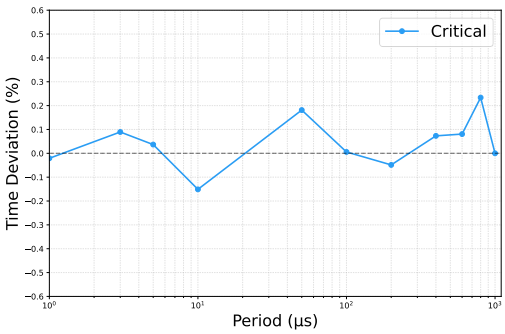


Figure 8 Time deviation on C.

Budget Configuration. Increasing the MBR period while keeping the budget fixed at 100 reduces the cache line write capacity of the non-critical, as shown in the Figure 5. For instance, with a period of 25 µs, cache line writes decrease by 2.6 times compared to a period of 1 µs. This outcome reflects how longer periods effectively limit the NC's access to memory bandwidth, as the delay in budget reset constrains the number of operations the non-critical can perform within each interval. These findings align with our expectation that increasing the period value reduces memory bandwidth availability for the non-critical, thus lowering its potential to interfere with the critical VM.

Period Configuration. Figure 6 demonstrates that with a fixed period of 10 µs, increasing the budget significantly increases the non-critical's cache line writes. For instance, a budget setting of 500 allows 2.2 times more cache line writes compared to a budget of 25. Higher budgets allocate more memory bandwidth to the non-critical VM, which in turn elevates its cumulative memory access capacity over time, maximizing the interference potential with the critical VM.

Overhead. Additionally, our empirical results show that *H-MBR* provides VM-level isolation with minimal overhead. To accurately assess the MBR mechanism's overhead, we deployed the baremetal VM in a solo setup, without the critical Linux VM. In this configuration, the PMU interrupt was disabled to prevent the baremetal from idling, as the PMU interrupt would otherwise trigger a negligible overhead due to the long idle wait times (which greatly exceed the combined interrupt latency and callback execution time). Consequently, the observed MBR overhead reflects a scenario where the budget never expires, and only the timer interrupt is active. As shown in Figure 7, the observed overhead for the non-critical VM remains minimal overall, with a noticeable spike only at a regulation period of 1 µs, where it reaches 14.3%. This elevated overhead is anticipated due to the frequent timer interrupts within such a short interval. However, as the regulation period increases slightly, the overhead rapidly diminishes: at 2 µs, it drops to 2%, and by 10 µs, it is already below 1%. For periods beyond 10 µs, the overhead becomes negligible, approaching zero. This trend demonstrates that the MBR mechanism maintains low interference impact on the non-critical VM, especially at moderate to higher regulation periods, reinforcing its effectiveness in providing controlled memory isolation.

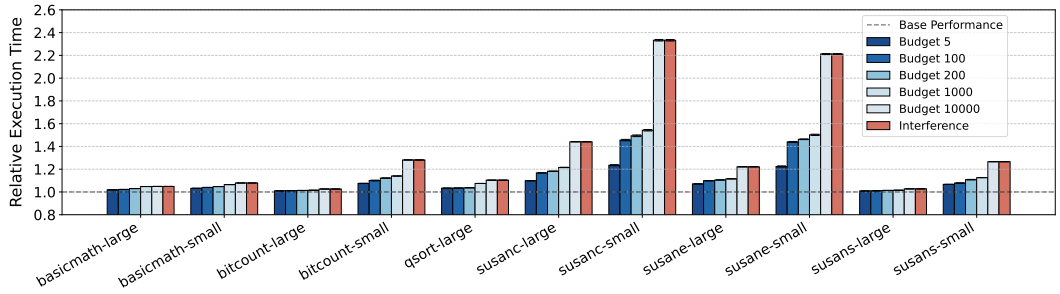


Figure 9 Fixed 10µs period.

4.2 Effect of MBR on Linux Benchmark Performance

This section presents results for the *interf+mbr* scenario, where MBR is applied to the non-critical baremetal VM, leaving the critical Linux VM unregulated. This approach enables a focused evaluation of MBR's impact on the critical VM's performance stability under varying MBR configurations. Two key aspects of MBR are analyzed: the influence of different budget values at a fixed period and the effect of varying periods at a fixed budget. These parameter variations directly affect memory bandwidth according to Bandwidth $= \frac{\text{Budget}}{\text{Period}}$.

Fixed Period, Variable Budgets. Figure 9 presents the effects of increasing the MBR budget for the non-critical VM with a fixed 10 µs period. Results show that as the MBR budget allocated to the non-critical VM increases, performance degradation in the critical VM also rises across all benchmarked workloads. This outcome aligns with expectations: a larger MBR budget allows the non-critical baremetal application increased memory bandwidth, thereby intensifying interference within shared memory resources. With budgets ranging from 50 to 10,000 memory accesses per period, lower budgets (e.g., a budget of 50) yield minimal interference, achieving performance levels close to the baseline for the critical VM. Higher budgets, such as 1,000 and 10,000, produce more substantial interference, notably in memory-intensive benchmarks like *susanc* and *susane*, where the relative execution time arises up to 2.3x and 2.2x, respectively. These benchmarks experience considerable performance degradation at higher budgets due to their sensitivity to memory contention. Interestingly, even with the short 10 µs period, small budgets (e.g., 100) still cause a measurable impact on the critical VM's performance, increasing the execution time by 1.45x for the susanc-small_benchmark.

Fixed Budget, Variable Periods. Figure 10 illustrates how different regulation periods, from 1 µs to 1000 µs, impact critical VM performance when the MBR budget for the non-critical VM is fixed at 100. Here, we observe that longer periods provide greater protection to the critical VM by reducing memory contention. This is especially evident in memory-sensitive applications, where the extended periods give the critical VM uninterrupted access to memory resources before the non-critical VM's budget resets, thus improving performance. For instance, with a 1 µs period, critical benchmarks like *susanc* show a performance increase of 1.78x compared to the solo execution. However, as the period extends to 1000 µs, performance approaches solo levels, indicating a substantial reduction in interference. These results underscore that longer periods limit the non-critical VM's memory bandwidth utilization, effectively mitigating its impact on the critical VM's workloads.

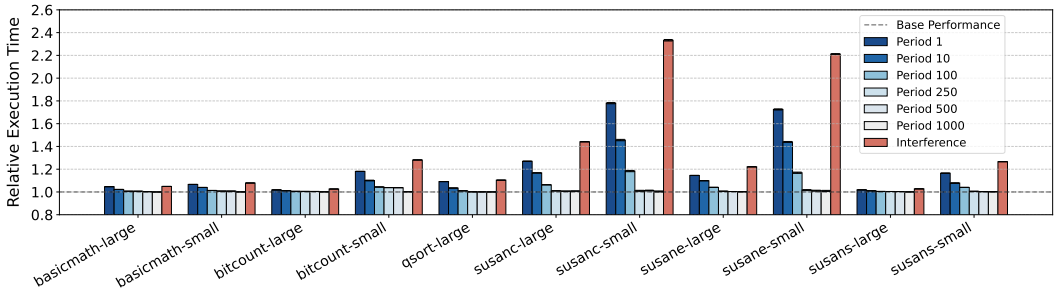


Figure 10 Fixed budget of 100.

Overhead. The results presented in Figure 8 confirm that applying MBR on the non-critical VM does not introduce any overhead on the critical VM, as each VM has dedicated, isolated CPUs. Additionally, the timer interrupts from MBR that regulate the non-critical VM's memory access do not interfere with the critical VM, resulting in effectively zero overhead. This is evidenced by the consistency in the critical VM's execution times across various MBR regulation periods, where variations remain minimal, within the range of 0.01% to 0.2%, equating to a minor fluctuation of around 20 µs. These findings underscore MBR's effectiveness for mixed-criticality systems, as it maintains reliable performance in the critical VM without impact from adjustments in the non-critical VM.

5 Discussion and Future Work

Currently, the MBR budget and period parameters are set statically at compile-time. Future work encompasses the introduction of dynamic configuration based on runtime monitoring of memory usage patterns could enable more adaptive and optimized bandwidth allocation. This could involve leveraging machine learning techniques to predict memory demands and proactively adjust MBR settings. Integration with additional isolation mechanisms, such as cache coloring, is another promising direction. By combining MBR with these techniques, contention could be mitigated at multiple levels of the memory hierarchy, providing even stronger temporal isolation guarantees. The interplay between these mechanisms and their cumulative impact on performance and predictability warrants further investigation. Benchmarking other guest OSs with MBR is also an important consideration. Since the mechanism targets real-time applications, and already supports other OSs, benchmarking a RTOS like Zephyr, would further enhance *H-MBR's* potential. Demonstrating the portability and generalizability of *H-MBR* would broaden its applicability across a wider range of use cases. Finally, long-term efforts could explore extending *H-MBR* beyond CPUs to heterogeneous computing elements like GPUs, FPGAs, and AI accelerators. As embedded systems increasingly incorporate these specialized components, managing their shared memory resources becomes critical. Adapting MBR mechanisms to these contexts could unlock new possibilities for predictable acceleration in mixed-criticality environments.

6 Conclusion

This work introduced *H-MBR*, a VM-centric MBR mechanism for MCS on multicore platforms. Through extensive evaluations, the mechanism demonstrated its effectiveness in mitigating memory contention and enhancing isolation between VMs, while it presented a remarkably low overhead. The evaluation results showed that *H-MBR* significantly reduced the interference on critical tasks by carefully controlling memory bandwidth reservation. Under configurations

where MBR budget was kept low, critical workloads showed minimal to no performance degradation. Overhead analysis further confirmed that *H-MBR* imposes minimal impact on workloads, even when dealing with minor periods. In the worst-case scenario of a 1 µs regulation period, the mechanism incurs an overhead of up to 14%; however, for regulation periods of 2 µs or longer, this overhead drops to below 1%. Compared to existing solutions, this is the lowest interrupt overhead observed due to Bao's optimized interrupt handling and lightweight design. This capability makes *H-MBR* stand apart from other implementations, specially in embedded applications, where it maximizes available processing resources for critical tasks without compromising real-time performance.

References

1 Jaume Abella, Carles Hernández, Eduardo Quiñones, Francisco Cazorla, Philippa Ryan, Mikel Azkarate-askasua, Jon Perez-Cerrolaza, Enrico Mezzetti, and Tullio Vardanega. WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *Proc. IEEE Int. Symp. on Industrial Embedded Systems*, 2015.

2 Ankit Agrawal, Renato Mancuso, Rodolfo Pellizzoni, and Gerhard Fohler. Analysis of Dynamic Memory Bandwidth Regulation in Multi-core Real-Time Systems. In *IEEE Real-Time Systems Symposium*, pages 230–241, 2018. doi:10.1109/RTSS.2018.00040.

3 Michael G Bechtel and Heechul Yun. Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention . In *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium* , 2019.

4 Gianluca Brilli, Roberto Cavicchioli, Marco Solieri, Paolo Valente, and Andrea Marongiu. Evaluating Controlled Memory Request Injection for Efficient Bandwidth Utilization and Predictable Execution in Heterogeneous SoCs. *ACM Transactions on Embedded Computing Systems*, 22:1–25, 2022. doi:10.1145/3548773.

5 Paolo Burgio, Marko Bertogna, Nicola Capodieci, Roberto Cavicchioli, Michal Sojka, Přemysl Houdek, Andrea Marongiu, Paolo Gai, Claudio Scordino, and Bruno Morelli. A software stack for next-generation automotive systems on many-core heterogeneous platforms. In *Microprocess. Microsyst.*, volume 52, 2017. doi:10.1016/J.MICPRO.2017.06.016.

6 Daniel Casini, Alessandro Biondi, Geoffrey Nelissen, and Giorgio Buttazzo. A Holistic Memory Contention Analysis for Parallel Real-Time Tasks under Partitioned Scheduling. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 239–252, 2020. doi:10.1109/RTAS48715.2020.000-3.

7 Francisco J. Cazorla, Leonidas Kosmidis, Enrico Mezzetti, Carles Hernandez, Jaume Abella, and Tullio Vardanega. Probabilistic Worst-Case Timing Analysis: Taxonomy and Comprehensive Survey. In *ACM Comput. Surv.*, volume 52, 2019. doi:10.1145/3301283.

8 Jon Perez Cerrolaza, Roman Obermaisser, Jaume Abella, Francisco J. Cazorla, Kim Grüttner, Irune Agirre, Hamidreza Ahmadian, and Imanol Allende. Multi-Core Devices for Safety-Critical Systems: A Survey. In *ACM Comput. Surv.*, volume 53(4), 2020.

9 Diogo Costa, Luca Cuomo, Daniel Oliveira, Ida Savino, Bruno Morelli, José Martins, Alessandro Biasci, and Sandro Pinto. IRQ Coloring and the Subtle Art of Mitigating Interrupt-Generated Interference. In *IEEE 29th International Conference on Embedded and Real-Time Computing Systems and Applications*, 2023.

10 Diogo Costa, Luca Cuomo, Daniel Oliveira, Ida Maria Savino, Bruno Morelli, José Martins, Fabrizio Tronci, Alessandro Biasci, and Sandro Pinto. IRQ Coloring: Mitigating Interrupt-Generated Interference on ARM Multicore Platforms. In *Fourth Workshop on Next Generation Real-Time Embedded Systems*, pages 1–13, 2023. doi:10.4230/OASICS.NG-RES.2023.2.

11 Alfons Crespo, Patricia Balbastre, Jose Simo, Javier Coronel, Daniel Gracia Pérez, and Philippe Bonnot. Hypervisor-Based Multicore Feedback Control of Mixed-Criticality Systems. In *IEEE Access*, volume 6, 2018.

12 Farzad Farshchi, Qijing Huang, and Heechul Yun. BRU: Bandwidth Regulation Unit for Real-Time Multicore Processors. In *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium*, 2020.

13 Marisol García-Valls, Tommaso Cucinotta, and Chenyang Lu. Challenges in Real-Time Virtualization and Predictable Cloud Computing. In *booktitle of Systems Architecture*, volume 60, pages 726–740, 2014. doi:10.1016/J.SYSARC.2014.07.004.

14 Angelo Garofalo, Manuele Rusci, Francesco Conti, Davide Rossi, and Luca Benini. PULP-NN: Accelerating Quantized Neural Networks on Parallel Ultra-Low-Power RISC-V Processors. In *Philosophical Transactions of the Royal Society A*, volume 378, pages 01–55, 2020.

15 Everaldo P. Gomes, Gabriel De O. Aguiar, and Giovani Gracioli. Implementation and Evaluation of MemGuard in the Bao Hypervisor. In *XIV Brazilian Symposium on Computing Systems Engineering*, pages 1–6, 2024. doi:10.1109/SBESC65055.2024.10771909.

16 Giovani Gracioli, Rohan Tabish, Renato Mancuso, Reza Mirosanlou, Rodolfo Pellizzoni, and Marco Caccamo. Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms. In *Proc. Euromicro Conf. Real-Time Syst.*, volume 133, 2019. doi:10.4230/LIPICS.ECRTS.2019.27.

17 M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proc. IEEE Int. Workshop on Workload Characterization*, pages 3–14, 2001.

18 Denis Hoornaert, Shahin Roozkhosh, and Renato Mancuso. A Memory Scheduling Infrastructure for Multi-Core Systems with Re-Programmable Logic. In *33rd Euromicro Conference on Real-Time Systems*, pages 1–22, 2021. doi:10.4230/LIPICS.ECRTS.2021.2.

19 Hyoseung Kim and Ragunathan (Raj) Rajkumar. Predictable Shared Cache Management for Multi-Core Real-Time Virtualization. In *ACM Trans. Embed. Comput. Syst.*, volume 17, 2017.

20 Tomasz Kloda, Marco Solieri, Renato Mancuso, Nicola Capodieci, Paolo Valente, and Marko Bertogna. Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 1–14, 2019. doi:10.1109/RTAS.2019.00009.

21 Liangzhen Lai, Naveen Suda, and Vikas Chandra. CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs. *arXiv preprint*, pages 1–10, 2018. arXiv:1801.06601.

22 Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. Real-time cache management framework for multicore architectures. In *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium*, 2013.

23 José Martins, Adriano Tavares, Marco Solieri, Marko Bertogna, and Sandro Pinto. Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems. In *Proc. Next Generation Real-Time Embedded Systems*, volume 77, 2020. doi:10.4230/OASICS.NG-RES.2020.3.

24 José Martins and Sandro Pinto. Shedding Light on Static Partitioning Hypervisors for Arm-based Mixed-Criticality Systems. In *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium*, 2023.

25 Paolo Modica, Alessandro Biondi, Giorgio Buttazzo, and Anup Patel. Supporting Temporal and Spatial Isolation in a Hypervisor for ARM Multicore Platforms. In *Proc. IEEE Int. Conf. on Industrial Technology*, 2018.

26 Afonso Oliveira, Gonçalo Moreira, Diogo Costa, Sandro Pinto, and Tiago Gomes. IA&AI: Interference Analysis in Multi-core Embedded AI Systems. In *International Conference on Data Science and Artificial Intelligence*, pages 181–193, 2024.

27 Marco Pagani, Enrico Rossi, Alessandro Biondi, Mauro Marinoni, Giuseppe Lipari, and Giorgio Buttazzo. A Bandwidth Reservation Mechanism for AXI-Based Hardware Accelerators on FPGAs. In *31st Euromicro Conference on Real-Time Systems*, pages 1–24, 2019. doi:10.4230/LIPICS.ECRTS.2019.24.

28 Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A Predictable Execution Model for COTS-Based Embedded Systems. In *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium*, 2013.

29 Sandro Pinto, Hugo Araújo, Daniel Oliveira, José Martins, and Adriano Tavares. Virtualization on TrustZone-enabled Microcontrollers? Voila! In *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium*, 2019.

30 Simon Reder and Jürgen Becker. Interference-Aware Memory Allocation for Real-Time Multi-Core Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2020.

31 Shahin Roozkhosh and Renato Mancuso. The Potential of Programmable Logic in the Middle: Cache Bleaching. In *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium* , 2020.

32 Bernd Schulz and Björn Annighöfer. Evaluation of Adaptive Partitioning and Real-Time Capability for Virtualization With Xen Hypervisor. *IEEE Transactions on Aerospace and Electronic Systems*, 58(1):206–217, 2021. doi:10.1109/TAES.2021.3104941.

33 Eric Seals, Michael Bechtel, and Heechul Yun. BandWatch: A System-Wide Memory Bandwidth Regulation System for Heterogeneous Multicore. In *IEEE 29th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 38–46, 2023.

34 Alejandro Serrano-Cases, Juan M. Reina, Jaume Abella, Enrico Mezzetti, and Francisco J. Cazorla. Leveraging Hardware QoS to Control Contention in the Xilinx Zynq UltraScale+ MPSoC. In *Proc. Euromicro Conf. Real-Time Syst.*, 2021.

35 Prateek Sharma, Purushottam Kulkarni, and Prashant Shenoy. Per-vm page cache partitioning for cloud computing platforms. In *2016 8th International Conference on Communication Systems and Networks*, pages 1–8, 2016. doi:10.1109/COMSNETS.2016.7439971.

36 Parul Sohal, Rohan Tabish, Ulrich Drepper, and Renato Mancuso. Profile-driven memory bandwidth management for accelerators and CPUs in QoS-enabled platforms. In *Real-Time Systems*, pages 235–274, 2022. doi:10.1007/S11241-022-09382-X.

37 Meng Xu, Robert Gifford, and Linh Thi Xuan Phan. Holistic multi-resource allocation for multicore real-time virtualization. In *Proceedings of the 56th Annual Design Automation Conference*, 2019.

38 Meng Xu, Linh Phan, Hyon-Young Choi, and Insup Lee. vCAT: Dynamic Cache Management Using CAT Virtualization. In *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium* , pages 211–222, 2017.

39 Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. COLORIS: A Dynamic Cache Partitioning System Using Page Coloring. In *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 381–392, 2014. doi:10.1145/2628071.2628104.

40 Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms. In *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium* , 2014.

41 Heechul Yun, Gang Yao, R. Pellizzoni, Marco Caccamo, and Lui Sha. MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-core Platforms. In *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium*, 2013.

42 Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory Access Control in Multiprocessor for Real-Time Systems with Mixed Criticality. In *Proc. Euromicro Conf. Real-Time Syst.* , 2012.

43 Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory Bandwidth Management for Efficient Performance Isolation in Multi-Core Platforms. In *IEEE Transactions on Computers*, volume 65(2), pages 562–576, 2016. doi:10.1109/TC.2015.2425889.

44 Matteo Zini, Giorgiomaria Cicero, Daniel Casini, and Alessandro Biondi. Profiling and controlling I/O-related memory contention in COTS heterogeneous platforms. In *Softw. Pract. Exper.*, volume 52(5), 2022. doi:10.1002/SPE.3053.

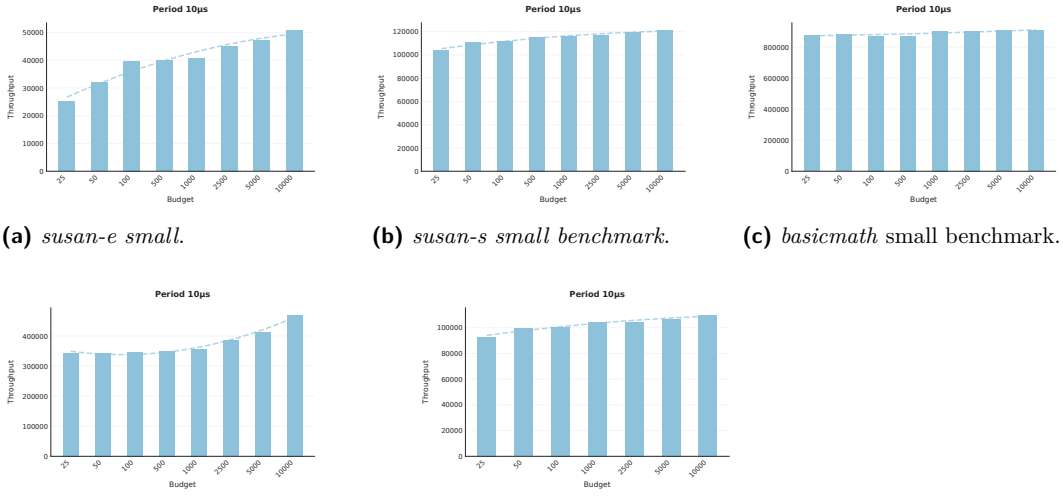
45 Alexander Zuepke, Andrea Bastoni, Weifan Chen, Marco Caccamo, and Renato Mancuso. MemPol: Policing Core Memory Bandwidth from Outside of the Cores. In *IEEE 29th Real-Time and Embedded Technology and Applications Symposium*, pages 235–248, 2023. doi:10.1109/RTAS58335.2023.00026.

A Appendix

Impact of MBR on Baremetal Throughput and Overhead

This appendix builds on the discussion in Section 4.1, which evaluated the memory-intensive *susan-c* benchmark running on the critical Linux-based VM (C) and focused on the throughput of the non-critical VM (NC). Here, we extend the analysis to additional benchmarks – *susan-e*, *susan-s*, *basicmath*, *qsort*, and *bitcount* – under the same conditions to provide a broader perspective on MBR's impact on different workloads. By examining these diverse benchmarks, we gain deeper insights into the effectiveness of MBR in managing memory contention across different types of computational tasks.

- ***Susan-e and Susan-s (Figures 11a and 11b):*** The NC demonstrated significant improvements in throughput as the budget increased. These results underscore the high memory bandwidth usage of both benchmarks.
- ***Basicmath (Figure 11c):*** Unlike the other benchmarks, the NC running alongside *basicmath* displayed minimal sensitivity to budget changes, maintaining consistently high throughput across all configurations. This stability underscores the benchmark's lower reliance on memory bandwidth for its computational tasks.
- ***Qsort (Figure 11d):*** The NC VM showed moderate throughput improvement as the budget increased, though not as dramatically as with the *susan* benchmarks, indicating moderate memory bandwidth dependency.
- ***Bitcount (Figure 11e):*** The NC VM's throughput showed moderate improvement with increasing budgets, reflecting the benchmark's medium memory bandwidth dependency, with a more gradual scaling compared to *susan-e* and *susan-s*.



(a) *susan-e small.* **(b)** *susan-s small benchmark.* **(c)** *basicmath* small benchmark.

(d) *qsort* small benchmark. **(e)** *bitcount* small benchmark.

Figure 11 Throughput variation on NC VM with different budget configurations for various benchmarks. Benchmarks exhibit different levels of sensitivity to budget changes based on their memory intensity.

These findings underscore the adaptability and practical impact of MBR in managing memory bandwidth effectively. Memory-intensive benchmarks such as *susan-e*, *susan-s*, and *bitcount* display noticeable improvements in NC throughput when allocated higher budgets,

reflecting their heavy reliance on memory resources. Conversely, memory-lighter benchmarks like *basicmath* remain largely unaffected, proving that they are less constrained by memory bandwidth. This demonstrates the importance of workload-specific MBR tuning: adjusting budgets based on workload demands can maximize system performance while preserving isolation.

SP-IMPact: A Framework for Static Partitioning Interference Mitigation and Performance Analysis

Diogo Costa
Centro ALGORITMI / LASI, Universidade do Minho, Portugal

Gonçalo Moreira
Centro ALGORITMI / LASI, Universidade do Minho, Portugal

Afonso Oliveira
Centro ALGORITMI / LASI, Universidade do Minho, Portugal

José Martins
Centro ALGORITMI / LASI, Universidade do Minho, Portugal

Sandro Pinto
Centro ALGORITMI / LASI, Universidade do Minho, Portugal

Abstract

Modern embedded systems are evolving toward complex, heterogeneous architectures to accommodate increasingly demanding applications. Driven by industry SWAP-C (Size, Weight, Power, and Cost) constraints, this shift has led to the consolidation of multiple systems onto single hardware platforms. Static Partitioning Hypervisors (SPHs) offer a promising solution to partition hardware resources and provide spatial isolation between critical workloads. However, shared hardware resources like the Last-Level Cache (LLC) and system bus can introduce significant temporal interference between virtual machines (VMs), negatively impacting performance and predictability. Over the past decade, academia and industry have focused on developing interference mitigation techniques, such as cache partitioning and memory bandwidth reservation. Configuring these techniques, however, is complex and time-consuming. Cache partitioning requires careful balancing of cache sections across VMs, while memory bandwidth reservation requires tuning bandwidth budgets and periods. With numerous possible configurations, testing all combinations is impractical and often leads to suboptimal configurations. Moreover, there is a gap in understanding how these techniques interact, as their combined use can result in compounded or conflicting effects on system performance. Static analysis solutions that estimate worst-case execution times (WCET) and upper bounds on execution times provide some guidance for configuring interference mitigation techniques. While useful in identifying potential interference effects, these tools often fail to capture the full complexity of modern multi-core systems, as they typically focus on a limited set of shared resources and neglect other sources of contention, such as IOMMUs and interrupt controllers. To address these challenges, we introduce SP-IMPact, an open-source framework designed to analyze and guide the configuration of interference mitigation techniques, through the deployment of diverse VM configurations and setups, and assessment of hardware-level contention (leveraging SPHs). It supports two mitigation techniques: (i) cache coloring and (ii) memory bandwidth reservation, while also evaluating the interactions between these techniques and their cumulative impact on system performance. By providing insights on real hardware platforms, SP-IMPact helps to optimize the configuration of these techniques in mixed-criticality systems, ensuring both performance and predictability.

2012 ACM Subject Classification Computer systems organization → Real-time system specification; Computer systems organization → Embedded software

Keywords and phrases Virtualization, Contention, Multi-core Interference, Mixed-Criticality Systems, Arm

Digital Object Identifier 10.4230/OASIcs.NG-RES.2025.5

Supplementary Material *Software (Source Code)*: `https://gitlab.com/ESRGv3/sp-impact`

Funding *Diogo Costa*: Supported by FCT grant 2022.13378.BD.
José Martins: Supported by FCT grant SFRH/BD/138660/2018.

1 Introduction

In recent decades, a significant trend toward digitization has revolutionized various industries including automotive, robotics, medical, and aerospace [9, 40, 41]. This shift brought an exponential increase in system features, prompting high-end embedded platforms to evolve from basic designs. Past simple MCUs with single cores have given way to today's intricate and highly complex platforms [11]. The transition from single-core to multi-core architectures, accommodating multiple CPUs, and integrating diverse hardware accelerators like Graphics Processing Units (GPUs), Tensor Processing Units (TPUs), Neural Processing Units (NPUs), and Field-Programmable Gate Arrays (FPGAs) [18, 33, 37], has fundamentally altered the landscape, resulting in highly heterogeneous designs.

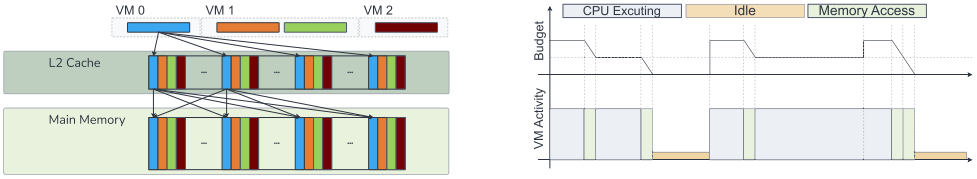
Simultaneously, market demands for compact and efficient systems have driven the consolidation of multiple functionalities onto single hardware platforms to meet Size, Weight, Power, and Cost (SWaP-C) constraints. This consolidation has led to the rise of Mixed Criticality Systems (MCSs) [21], where components with varying criticality levels coexist on the same platform. Virtualization technologies have been instrumental in enabling such consolidation, with hypervisor-based solutions – particularly static-partitioning hypervisors [35, 34, 41, 22, 27] – striking a balance between safety, security, and resource efficiency. These hypervisors allow for the deployment of diverse workloads within MCSs while adhering to stringent industry safety standards, such as ISO 26262 [39].

Achieving robust system consolidation requires addressing critical challenges to ensure safety and security, particularly spatial and temporal isolation. Spatial isolation guarantees that architectural resources (e.g., CPUs and main memory) allocated to one system remain inaccessible to others. Temporal isolation ensures that the execution of one system's workloads does not interfere with another's timing requirements. While static partitioning effectively addresses spatial isolation, temporal isolation remains a significant challenge due to contention on shared microarchitectural resources like the Last-Level Cache (LLC), main memory, and system bus. Such contention leads to increased execution times and reduced determinism [1, 7, 8, 31, 1], making timing predictability particularly difficult for hard real-time systems.

Techniques such as cache partitioning [17] and memory bandwidth reservation [51] have emerged as promising solutions to mitigate temporal interference in MCSs. Cache partitioning segments the LLC into regions assigned to specific Virtual Machines (VMs), while memory bandwidth reservation regulates the number of memory accesses within a given time frame. However, configuring these techniques effectively requires careful balancing of resources across VMs and fine-tuning parameters (e.g., define cache regions and/or memory budgets and periods). This process is complex, time-consuming, and impractical for real-world MCSs, often leading to suboptimal configurations. Static analysis tools [1, 3] have been explored to address these challenges, offering a means to understand interference impacts in MCSs and guide the configuration of mitigation techniques. By estimating Worst-Case Execution Time (WCET) and quantifying interference effects, these tools provide a foundation for informed decision-making. However, existing static analysis solutions often focus on specific shared resources, such as the LLC, overlooking other shared hardware resources (e.g., IOMMUs and interrupt controllers).

To address these limitations, we introduce SP-IMPact, an open-source framework[1] designed to analyze and support the configuration of interference mitigation techniques. SP-IMPact enables a comprehensive understanding of the impact of shared hardware resources

[1] `https://gitlab.com/ESRGv3/sp-impact`



(a) Cache Coloring Toy Example. **(b)** Memory Bandwidth Reservation Toy Example.

Figure 1 Illustrative examples of cache coloring and memory bandwidth reservation mechanisms.

on real platforms by considering all potential sources of contention - filling critical gaps left by currently available solutions. Furthermore, it allows the deployment of diverse configurations of cache coloring and memory bandwidth reservation to evaluate their effects on the workloads of different VMs. The insights gained through SP-IMPact can guide the optimization of these techniques, easing their usage by industry due to the framework's workload-agnostic design, which supports various operating systems and workloads. For academia, SP-IMPact provides a versatile launchpad for deploying and testing new interference mitigation techniques. While this paper leverages the Bao hypervisor [34] as a use case, SP-IMPact is agnostic to the underlying hypervisor and can be extended to support other static partitioning hypervisors.

2 Background

The consolidation of MCSs introduces a well-known challenge: interference between co-existing VMs, which can degrade performance and disrupt real-time guarantees [18, 34, 28, 50, 48, 51, 4, 46, 13, 30, 29]. This interference typically arises from contention over shared micro-architectural resources, such as the LLC, main memory, and the system bus, leading to increased execution time and lack of determinism. To mitigate these issues, techniques such as cache partitioning [28, 36, 26] and memory bandwidth reservation [51, 50, 36, 15], among others solutions targeting I/O and interrupts regulation [52, 16, 12] , have been proposed.

Cache Partitioning. Cache partitioning techniques enable the selective allocation of cache regions to specific workloads, thereby reducing cache contention and improving predictability. One of the most used techniques for cache partitioning is cache coloring, which divides the LLC into distinct regions by assigning specific "colors" to individual workloads, where each "color" corresponds to different cache sets, which helps to control cache access patterns and minimize interference. Cache coloring is commonly used to assign dedicated portions of the cache to individual VMs, effectively limiting LLC contention, as depicted in Figure 1 (a).

Memory Bandwidth Reservation. Memory bandwidth reservation techniques regulate the memory access rate to reduce interference and ensure temporal isolation. MemGuard [51], for example, limits the memory access rate per CPU by allocating specific portions of memory bandwidth, or "budgets", over defined periods, as depicted in Figure 1 (b). This prevents any workload from monopolizing memory, ensuring fair resource distribution and reducing delays, thus improving predictable performance and minimizing interference between workloads.

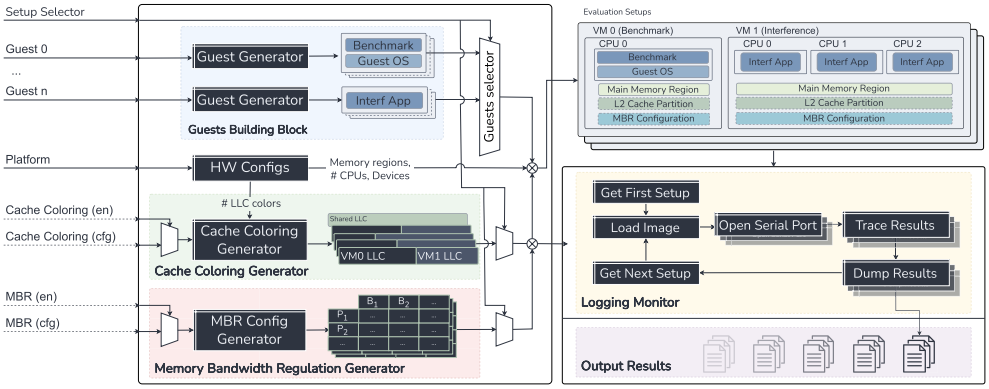


Figure 2 SP-IMPact System Overview.

3 System Overview

In this paper, we introduce SP-IMPact, a framework developed to evaluate and benchmark the performance of MCSs, with a focus on measuring interference and assessing the effectiveness and interaction of interference mitigation techniques. Using a configuration file, users can specify the platform, guest definitions, and test setups. As shown in Figure 2, the framework leverages three key components: (i) the **Guest Generator**, which build guests; (ii) the **Cache Coloring Generator**, which defines cache partitioning configurations; and the (iii) **Memory Bandwidth Regulation Generator**, which generates different configurations of the mechanism. Together, these components ensure precise and consistent performance evaluations. SP-IMPact also includes a **Logging Monitor** for collecting run-time data and an **Output Results** module to handle the gathered information for further analysis.

Guests Generator. In the context of evaluating system performance, the framework provides support for constructing different types of guests, each tailored for specific benchmarking or interference generation tasks. For the sake of simplicity, this discussion focuses on two primary guest types: a Linux guest and a baremetal guest. However, it should be noted that the framework can be extended to support more and varied guest types as needed.

1. **Linux Benchmark**: Designed to simplify the deployment of various Linux-based workloads, enabling the evaluation of system behavior across diverse scenarios.
2. **Contention Engine**: A baremetal guest tailored to create memory and hardware resource pressure, targeting the LLC and main memory. Key parameters include CPU count, workload sizes, and operation types (reads, writes, or both).

To formalize these configuration options for the baremetal guest, let $\boldsymbol{M}$ represent the total number of CPUs available, $\boldsymbol{L}$ denote the set of possible cache line sizes, $\boldsymbol{W}$ signify the set of workload sizes, and $\boldsymbol{O}$ define the set of operation types, where $\boldsymbol{O} = \{\text{read}, \text{write}, \text{read/write}\}$. The configuration space for the baremetal guest can thus be expressed as:

$$G_{\text{baremetal}} = \{(m, l, w, o) \mid m \in M, l \in L, w \in W, o \in O\} \tag{1}$$

where $G_{\text{baremetal}}$ represents the set of all possible configurations for the baremetal guest. To represent the complete configuration space for all guests, including both baremetal and Linux guests, let G denote the overall set of guest configurations, which is equal to $G_{\text{baremetal}} \cup G_{\text{linux}}$, where G_{linux} represents the set of configurations for the Linux guest.

Cache Coloring Generator. In MCSs, cache partitioning is crucial for reducing LLC interference and ensuring predictable performance. This process, tipicaly based on cache coloring, aims to divide cache sets into non-overlapping regions for each VM. To assess the impact of different cache partitions, the BaoRTI framework supports the generation of distinct cache color configurations based on the following parameters:

1. The total number of cache sets $\boldsymbol{S}$, which corresponds to the bit length of the bitmap used to define the color assignments (i.e., each cache set index is represented as a bit in the range $[0, S-1]$);
2. The number of VMs, $\boldsymbol{N}$, to which distinct cache partitions will be assigned.

The objective of the function is to generate unique configurations of bit masks, dividing the bit range $[0, S-1]$ into $\boldsymbol{N}$ distinct non-overlapping sections. Each section represents a cache partition that can be assigned as a color to the VMs. Given $\boldsymbol{S}$ cache sets and $\boldsymbol{N}$ VMs, the function produces a unique configuration of non-overlapping bit masks for each VM. To facilitate this process, we denote $\boldsymbol{C}$ as the set of all possible $(N-1)$-combinations of bit positions within the range $[0, S-1]$. Each combination in $\boldsymbol{C}$ represents potential VM coloring configurations and can be formally defined as:

$$\boldsymbol{C} = \{(b_1, b_2, \ldots, b_{N-1}) \mid 0 \leq b_1 < b_2 < \cdots < b_{N-1} < S\} \tag{2}$$

where each b_i denotes a bit position that separates the partitions for each VM.

To generate the bit masks for each VM, we begin by initializing the starting bit s_0 to 0. For each VM i, where i ranges from 0 to $N-1$, we define the end bit e_i based on the boundary positions: if $i < N-1$, e_i is set to b_i, whereas for the last VM ($i = N-1$), e_i is assigned the total number of cache sets S. With these boundaries established, we compute the bit mask M_i for each VM using the formula:

$$M_i = ((1 \ll (e_i - s_i)) - 1) \ll s_i \tag{3}$$

where "$\ll$" represents a left bit shift operation. This formula creates a mask with $(e_i - s_i)$ bits set to 1, aligned to begin at the position defined by s_i. After calculating the bit mask, we update the starting bit s_i for the next VM by setting it to the current end bit e_i. This iterative process continues until all masks for the VMs are generated, ensuring that each VM receives a unique configuration of non-overlapping cache partitions. After generating a list of bit masks for each VM in the current combination, this configuration is added to a result set colors_assignments if it does not already exist in the set. This ensures that all configurations in that list are unique.

Memory Bandwidth Regulation Generator. In real-time systems, generating distinct memory bandwidth configurations for VMs is essential for ensuring predictable performance and efficient resource utilization. This process, known as memory bandwidth reservation, focuses on creating unique combinations of budget and sampling period for each VM. To evaluate the effects of various bandwidth configurations, the BaoRTI framework supports the generation of distinct MBR configurations based on the following parameters:

1. A list of budgets $\boldsymbol{B}$ available for reservation, where each budget specifies the maximum amount of bandwidth allocated to a VM.
2. A list of sampling periods $\boldsymbol{P}$, which define the time intervals at which the allocated bandwidth should be monitored.

The objective of the memory bandwidth reservation assignment generation is to produce all the possible combinations of budget and period assignments for each guest. Given the set of all guests, $\boldsymbol{G}$, where each guest, $\boldsymbol{g}$, is associated with a set of budgets, B_g, and a set of periods, P_g, the configuration for each guest can be expressed as:

$$MBR_g = \{(B, P) \mid B \in B_g, P \in P_g\} \tag{4}$$

where MBR_g represents the set of all combinations of memory bandwidth reservation configurations for the guest g. Let G be the total number of guests, B be the maximum number of budgets across guests, and P be the maximum number of sampling periods across guests. The time complexity for generating the budget-period combinations for each guest is $O(B \times P)$. Since there are G guests, the overall complexity for processing all guests and generating their combinations can be expressed as $O\left(G \times B \times P\right)$.

SP-IMPact features a results logging system designed to capture essential performance and behavioral metrics from the target platform during test execution. The framework collects data from multiple serial ports, each mapped to a specific \ac{VM}, ensuring comprehensive monitoring across the system. The captured metrics include execution time and key micro-architectural events, such as acLLC misses, memory access counts, and cycles spent on the system bus. These metrics are vital for evaluating the impact of shared hardware resources – like the LLC and memory controllers – on workload performance and predictability. This versatile design enables SP-IMPact to support a wide array of benchmarks and metrics tailored to diverse interference scenarios. By correlating data across multiple VMs and configurations, SP-IMPact provides the granularity required to assess the effectiveness of interference mitigation techniques and optimize their configurations.

4 Evaluation

4.1 Evaluation Setup

Hardware Platform. The experiments were conducted on a Xilinx ZCU104 evaluation board equipped with a Zynq Ultrascale+ ZU7EV SoC. This platform includes a quad-core Arm Cortex-A53 processor, operating at 1.2 GHz. While the SoC supports up to 16 distinct cache colors for cache coloring, the Bao hypervisor constrains this to 8 colors to avoid partitioning the L1 cache. Each core has a dedicated 32 KiB L1 instruction and data cache, along with a unified 1 MiB L2 cache. Additionally, the board is equipped with an Arm Performance Monitoring Unit (PMU), which was leveraged to collect microarchitectural events (such as cache misses and system bus accesses) and profile the benchmark.

Workloads. For our evaluation, we leveraged the MiBench Automotive and Industrial Control System (AICS) [19] Suite within the critical VM. This subset includes three memory-intensive benchmarks: *qsort*, *susan-c*, and *susan-e*. To generate interference at the memory hierarchy, we deployed a baremetal application that continuously performs read or write operations on a buffer with different sizes. Specifically, buffer sizes include 32 KiB (100% of the L1 cache), 512 KiB (50% of the L2 cache), 1 MiB (100% of the L2 cache), 1.5 MiB (150% of the L2 cache), 2 MiB (200% of the L2 cache), and 4 MiB (400% of the L2 cache).

Setups. According to Equation 1, we consider the following parameters: $L = 1$ (using only the cache line size matching the cache line size of the target hardware platform), $C = 1$ (using only one CPU configuration, which assigns 3 CPUs to the baremetal VM), $W = 6$ (the total

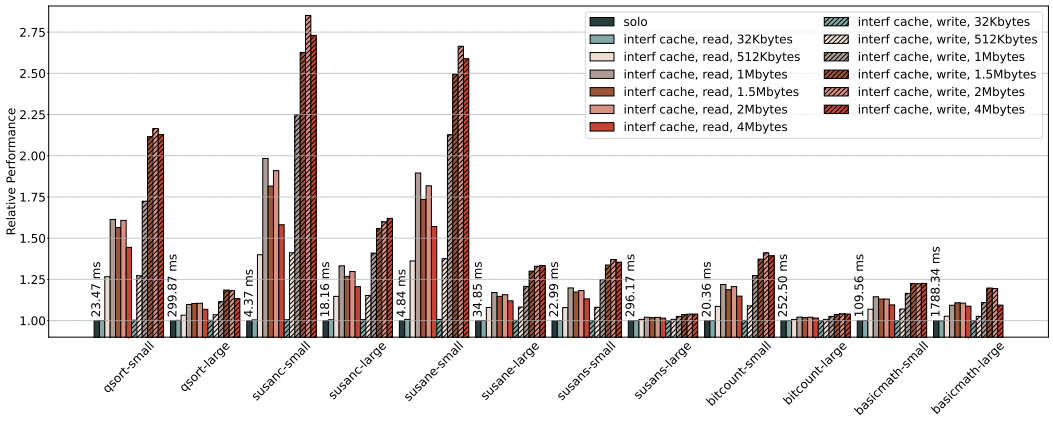
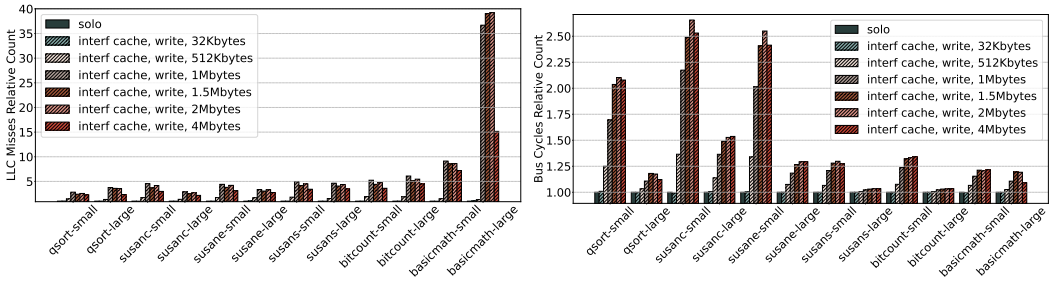


Figure 3 Performance overheads of MiBench automotive benchmark with different workloads.



(a) Interference Impact on LLC. **(b)** Interference Impact on system bus.

Figure 4 Collected PMU events from MiBench benchmark.

number of workloads used), and $O = 2$ (representing both read and write operations). This results in a total of 12 variations of the baremetal guest. Additionally, for cache coloring, since there are 2 VMs ($N = 2$) and 8 possible cache sets ($S = 8$), there are 8 unique configurations of cache coloring. However, we excluded scenarios in which the Linux VM would be allocated only a single cache color, as such configurations would not provide meaningful performance benefits. Thus, combining these factors results in a total of 84 setups to be tested. For simplicity, we will not consider the configuration of MBR, as introducing it would significantly increase the total number of setups in this evaluation section.

Setup Naming Convention. Setups are named `solo` or `interf_<access>_<buffer_size>`. The `solo` setup serves as the baseline, where a Linux VM runs the MiBench benchmarks without interference. In `interf_<access>_<buffer size>` setups, an additional workload creates cache contention, with `access` specifying `read` or `write` interference type and `buffer_size` indicating the buffer size used. Cache coloring setups add the suffix `<cc_num-colors>`, where `num-colors` denotes the cache colors allocated to the critical VM.

4.2 Interference Impact on Multi-core Platforms

Empirical results presented in Figure 3 indicate that contenetion on shared hardware resources can severely hamper the performance of memory-intensive benchmarks such as *qsort-small*, *susan-c-small*, and *susan-e-small*. The results confirm the theoretical expectations of how the interference buffer size influences resource contention, providing valuable insight into

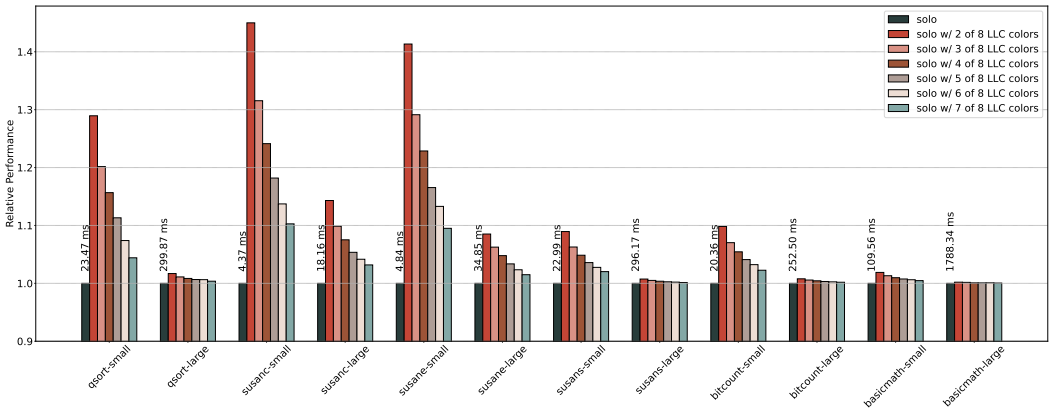
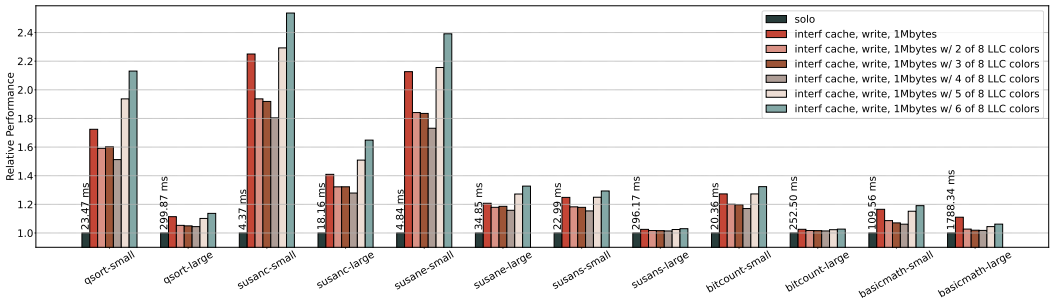


Figure 5 Cache coloring configuration impact on MiBench Benchmark.

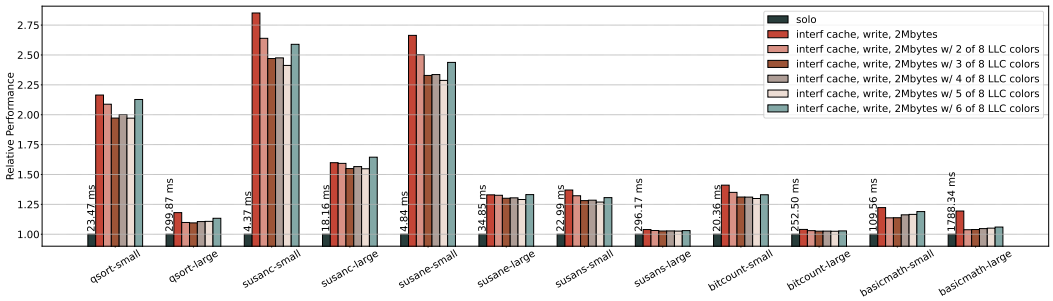
the SP-IMPact framework's role in identifying and quantifying such issues. This framework proves essential in assessing how system configurations can exacerbate or mitigate performance bottlenecks in multi-core platforms. The observed interference patterns, where larger buffer sizes lead to increased contention for shared resources, underscore the importance of understanding system-level interactions in MCSs.

While the the empirical resutls where theoretically expected, the empirical evidence reinforces the critical need for tools like SP-IMPact to understand the impact of consolidating different workloads on top of the same hardware platform. Not only does the framework help identify these issues, but it also enables developers to quantify the effects of interference under different configurations, a key insight to drive the deployment of MCSs. By running different workloads with different configurations, developers can collect key performance metrics, such as execution time, cache misses, and bus cycles, which are essential for understanding the severity of the interference. These metrics provide a comprehensive view of how shared resources (e.g. the LLC and the system bus) impact overall system performance. For example, as depicted in Figure 4(a), increasing the buffer size leads to a notable rise in cache misses, which in turn increases the execution time. Specifically, in the `interf_write_1MiB` scenario, the execution time for the *susanc-small* benchmark increases from 4.37 ms to 9.83 ms, demonstrating the growing impact of interference as the buffer size increases.

The role of the SP-IMPact framework in identifying these performance impacts is critical, as it helps pinpoint where interference is most pronounced. Once performance metrics are gathered, the framework allows for in-depth analysis to identify the root causes of performance degradation. For instance, as the interference buffer size grows, portions of the L2 cache become occupied, leading to cache contention and cache evictions. These evictions result in increased memory access time, contributing to further performance slowdowns. The underlying mechanism driving this issue is the competition for cache lines, which causes more frequent evictions and delays in data retrieval. This phenomenon is compounded by the finite size of the cache, which limits the amount of data that can be stored and retrieved quickly. Additionally, Figure 4(b) shows that increasing the buffer size also introduces contention on the system bus, further exacerbating the performance overhead. As workloads compete for access to shared bus resources, the time spent transferring data between the CPU and memory increases, leading to a marked decline in overall system efficiency. These findings underscore the importance of managing resource contention in multi-core environments, where shared hardware resources are increasingly stressed by demanding workloads.



(a) Cache Coloring on 1MiB interference scenario.



(b) Cache Coloring on 2MiB interference scenario.

Figure 6 Cache Coloring interference mitigation on MiBench Benchmark.

4.3 Interference Mitigation Techniques

Cache Coloring Overhead. Empirical results shows that cache coloring, even in a single-VM environment without interference, can introduce variations in benchmark performance depending on the number of cache colors available for the benchmark's use. This effect is most pronounced in memory-intensive benchmarks, such as *qsort-small*, *susan-c-small*, and *susan-e-small*, where reduced cache availability leads to significant slowdowns due to increased cache misses. With only two of the eight cache colors available, the execution time of benchmarks like *susanc-small* and *susane-small* increases by 1.45x and 1.41x, respectively, due to reduced cache allocation. As the number of colors increases, performance gradually approaches the baseline. With five cache colors available, benchmarks generally perform closer to their solo execution times. For example, *susanc-small* and *susane-small* improve to slowdowns of 1.18x and 1.17x, respectively. Benchmarks with lower memory intensity, such as *qsort-large*, *basicmath-small*, and *basicmath-large*, show minimal to no performance degradation across various cache coloring scenarios. With only two colors, *basicmath-large* shows no measurable slowdown across all coloring configurations. Similarly, *qsort-large* and *basicmath-small* maintain near-baseline performance, with minimal slowdowns of 1.02x.

Interference Mitigation. The SP-IMPact framework plays a key role in assisting developers with mitigating memory contention issues during the development of MCSs. After identifying bottlenecks caused by shared hardware resources, developers can leverage the framework to simulate different scenarios and adjust system configurations accordingly. For example, cache coloring can be leveraged to minimize interference. Figures 6(a) and 6(b) show the impact of different cache coloring configurations on memory-intensive benchmarks, such as *qsort-small* and *susanc-small*, when consolidated with the interference baremetal VM

(e.g, running the `interf_write_1MiB` and the `interf_write_2MiB` scenarios). Applying 2 cache colors reduces execution time overhead from 1.72x and 2.25x (no coloring) to 1.59x and 1.94x, respectively. Further improvements are observed with 4 cache colors, reducing interference to 1.51x and 1.80x for *qsort-small* and *susanc-small*. While cache coloring is effective for memory-intensive workloads, developers should consider diminishing returns beyond 4 colors, where performance gains decrease, and system-level contention (especially on the bus) may increase. The SP-IMPact framework helps identify these diminishing returns, allowing developers to select the most optimal configuration. For less memory-intensive benchmarks like *qsort-large* and *basicmath-large*, cache coloring has minimal impact, enabling developers to focus on other optimization techniques for such workloads.

5 Discussion and Future Directions

In this section, we discuss some of the open issues and potential research directions to understand and improve the impact of interference in multi-core platforms.

Workload Interference Analysis in Mixed-Criticality Systems. MCSs face significant challenges when consolidating workloads with varying criticality levels and timing requirements on the same hardware platform. As workloads compete for shared resources such as caches, memory buses, and system interconnects, predicting interactions and maintaining reliable performance for critical tasks remains a complex problem. While the framework presented in this work enables profiling and quantifying interference effects under diverse scenarios, further research is needed to explore how workload characteristics - such as memory access patterns and computation intensity (e.g., memory access rate - can be modeled more accurately. Moreover, one important limitation of the current evaluation is that it primarily focuses on interference effects in terms of LLC misses and bus cycles; while the SP-IMPact framework allows the analysis of the contention in these components, the lack of state-of-the-art benchmarks to evaluate them limits their inclusion in this study.

Interference Mitigation Techniques. Configuring interference mitigation mechanisms, such as cache coloring, presents its own set of challenges. Each possible configuration (e.g. the number of cache colors or memory bandwidth regulation configuration) can produce different impacts on performance and contention levels. Selecting the optimal configuration requires an understanding of both the workload's memory demands and the system's architectural characteristics. The framework aids in this process by providing in-depth evaluations of various interference mitigation techniques configurations and their impact on interference on multi-core platforms. Additionally, exploring the interaction between the proposed framework and high-performance hardware features, such as quality-of-service (QoS) mechanisms that control on-chip and DRAM traffic, would be valuable, as presented in [45].

Future Work. Building on the insights from this study, several extensions to the current framework are planned: an immediate enhancement of SP-IMPact involves the development of a hypervisor-level performance monitor to enable VM profiling without requiring guest instrumentation. Currently, the framework simplifies the generation of Linux-based benchmarks and baremetal VMs; next-steps focus on extending this capability to other OSes (e.g., FreeRTOS and Zephyr), enabling a comprehensive interference analysis and mitigation evaluations across a wider range of workloads and system configurations. In the long term, we aim

to integrate AI-driven techniques for adaptive interference management, which may enable the optimization of interference patterns, allowing the simulation of worst-case scenarios and providing more accurate performance assessments under challenging conditions.

6 Related Work

Interference analysis in multi-core systems has been approached using two primary frameworks: (i) generic task models and (ii) phased execution models. Each approach has its strengths but also limitations when it comes to capturing the complexities of modern high-complexity MCSs, especially those that include hypervisors. In the following, we provide a brief overview of these two categories and highlight the key research efforts within each.

Generic Task Models. Generic task models provide abstractions for task behaviors on multi-core systems by focusing on resource usage patterns such as memory access, computation, and synchronization. These models typically focus on quantifying contention between tasks based on broad assumptions and often omit platform-specific characteristics. Generic task models can be divided into two main categories: (i) memory bus contention and [10, 2, 42, 24, 23, 14], (ii) main memory contention [49, 25, 20].

Phased Execution Models. While generic task models provide broad abstractions, they are limited in their ability to model complex, dynamic interference patterns that arise in multi-core systems. This limitation led to the development of phased execution models, which break down task execution into distinct phases. Phased execution models offer more detailed representations of how tasks interact with shared resources during different execution phases. To address the issues left by generic task models, phased execution models are divided in: (i) offline scheduling-based approaches [44, 38, 5], (ii) shared resource contention-based approaches [32, 6, 3], and (iii) memory-centric scheduling-based approaches [47, 43].

Limitations of Existing Approaches. While generic task and phased execution models help understand some aspects of interference, they fall short in high-complexity MCSs, especially those with hypervisors. These models focus on limited contention sources, like LLC and system buses, but omit others such as IOMMUs or interrupt controllers, which are crucial in real hardware. Moreover, they overlook the combined effects of multiple mitigation techniques. SP-IMPact fills these gaps by enabling the assessment of interference in hypervisor-based systems and evaluating the effectiveness and interactions of interference mitigation techniques. Unlike analytical models, SP-IMPact simplifies configuration by allowing real-time experimentation on actual hardware, making it easier to identify bottlenecks and test configurations in a flexible way. The framework was tested with Bao hypervisor and currently supports its configuration interface to define VMs' configurations and hardware partitioning, but it can be extended to support other hypervisors in the future.

7 Conclusion

In this paper, we propose the design, implementation, and evaluation of SP-IMPact, a framework for analyzing the impact of multi-core contention, and the impact of interference mitigation techniques. This framework facilitates the automated deployment, configuration, and data collection of multiple setups to quantify platform-level contention and evaluate the impact of interference mitigation techniques, such as cache coloring. Using the Zynq Ultrascale+ platform, our evaluation demonstrated how the framework enables precise

analysis of interference effects under various workload configurations, providing critical insights for deploying consolidated multi-core systems. We believe that this framework lays a solid foundation for future extensions, including AI-driven interference management, expanded workload patterns, and support for additional platforms.

References

1 Jaume Abella, Carles Hernandez, Eduardo Quiñones, Francisco J. Cazorla, Philippa Ryan Conmy, Mikel Azkarate-askasua, Jon Perez, Enrico Mezzetti, and Tullio Vardanega. WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, 2015.

2 Alexandru Andrei, Zebo Peng, Jakob Rosen, and Petru Eles. Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip . In *IEEE 34th Real-Time Systems Symposium*, pages 49–60, 2007. doi:10.1109/RTSS.2007.24.

3 Jatin Arora, Cláudio Maia, Syed Aftab Rashid, Geoffrey Nelissen, and Eduardo Tovar. Bus-contention aware wcrt analysis for the 3-phase task model considering a work-conserving bus arbitration scheme. *Journal of Systems Architecture*, 122:102345, 2022. doi:10.1016/J.SYSARC.2021.102345.

4 Michael Bechtel and Heechul Yun. Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention . In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 357–367, 2019.

5 Matthias Becker, Dakshina Dasari, Borislav Nicolic, Benny Akesson, Vincent Nelis, and Thomas Nolte. Contention-Free Execution of Automotive Applications on a Clustered Many-Core Platform . In *28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 14–24, 2016.

6 Daniel Casini, Alessandro Biondi, Geoffrey Nelissen, and Giorgio Buttazzo. A Holistic Memory Contention Analysis for Parallel Real-Time Tasks under Partitioned Scheduling . In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 239–252, 2020. doi:10.1109/RTAS48715.2020.000-3.

7 Francisco J. Cazorla, Leonidas Kosmidis, Enrico Mezzetti, Carles Hernandez, Jaume Abella, and Tullio Vardanega. Probabilistic worst-case timing analysis: Taxonomy and comprehensive survey. *ACM Comput. Surv.*, 52, 2019. doi:10.1145/3301283.

8 Francisco J. Cazorla, Eduardo Quiñones, Tullio Vardanega, Liliana Cucu, Benoit Triquet, Guillem Bernat, Emery Berger, Jaume Abella, Franck Wartel, Michael Houston, Luca Santinelli, Leonidas Kosmidis, Code Lo, and Dorin Maxim. PROARTIS: Probabilistically Analyzable Real-Time Systems. *ACM Trans. Embed. Comput. Syst.*, 12(2s), 2013. doi:10.1145/2465787.2465796.

9 Jon Perez Cerrolaza, Roman Obermaisser, Jaume Abella, Francisco J. Cazorla, Kim Grüttner, Irune Agirre, Hamidreza Ahmadian, and Imanol Allende. Multi-core Devices for Safety-critical Systems: A Survey. *ACM Comput. Surv.*, 53(4), 2020. doi:10.1145/3398665.

10 Sudipta Chattopadhyay, Abhik Roychoudhury, and Tulika Mitra. Modeling shared cache and bus in multi-cores for timing analysis. In *Proceedings of the 13th international workshop on software & compilers for embedded systems*, pages 1–10, 2010.

11 Diogo Costa, Luca Cuomo, Daniel Oliveira, Ida Maria Savino, Bruno Morelli, José Martins, Fabrizio Tronci, Alessandro Biasci, and Sandro Pinto. IRQ Coloring: Mitigating Interrupt-Generated Interference on ARM Multicore Platforms. In *Fourth Workshop on Next Generation Real-Time Embedded Systems (NG-RES)*, volume 108, pages 2:1–2:13, 2023. doi:10.4230/OASICS.NG-RES.2023.2.

12 Diogo Costa, Luca Cuomo, Daniel Oliveira, Ida Maria Savino, Bruno Morelli, José Martins, Alessandro Biasci, and Sandro Pinto. IRQ Coloring and the Subtle Art of Mitigating Interrupt-Generated Interference. In *IEEE 29th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 47–56, 2023. doi:10.1109/RTCSA58653.2023.00015.

13 Dakshina Dasari, Benny Akesson, Vincent Nelis, Muhammad Ali Awan, and Stefan M Petters. Identifying the sources of unpredictability in cots-based multicore systems. In *8th IEEE international symposium on industrial embedded systems (SIES)*, pages 39–48, 2013.

14 Robert I Davis, Sebastian Altmeyer, Leandro S Indrusiak, Claire Maiza, Vincent Nelis, and Jan Reineke. An extensible framework for multicore response time analysis. *Real-Time Systems*, 54:607–661, 2018. doi:10.1007/S11241-017-9285-4.

15 Farzad Farshchi, Qijing Huang, and Heechul Yun. BRU: Bandwidth Regulation Unit for Real-Time Multicore Processors. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 364–375, 2020. doi:10.1109/RTAS48715.2020.00011.

16 Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. Eli: bare-metal performance for i/o virtualization. *SIGPLAN Not.*, 47(4):411–422, 2012. doi:10.1145/2150976.2151020.

17 Giovani Gracioli, Ahmed Alhammad, Renato Mancuso, Antônio Augusto Fröhlich, and Rodolfo Pellizzoni. A Survey on Cache Management Mechanisms for Real-Time Embedded Systems. *ACM Comput. Surv.*, 48(2), 2015. doi:10.1145/2830555.

18 Giovani Gracioli, Rohan Tabish, Renato Mancuso, Reza Mirosanlou, Rodolfo Pellizzoni, and Marco Caccamo. Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms. In *31st Euromicro Conference on Real-Time Systems (ECRTS)*, volume 133, pages 27:1–27:25, 2019. doi:10.4230/LIPICS.ECRTS.2019.27.

19 M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization.*, pages 3–14, 2001.

20 Mohamed Hassan and Rodolfo Pellizzoni. Bounding DRAM Interference in COTS Heterogeneous MPSoCs for Mixed Criticality Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2323–2336, 2018. doi:10.1109/TCAD.2018.2857379.

21 Thomas A. Henzinger and Joseph Sifakis. "The Embedded Systems Design Challenge". In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*, pages 1–15, 2006. doi:10.1007/11813040_1.

22 Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones. In *5th IEEE Consumer Communications and Networking Conference*, pages 257–261, 2008. doi:10.1109/CCNC08.2007.64.

23 Michael Jacobs, Sebastian Hahn, and Sebastian Hack. A Framework for the Derivation of WCET Analyses for Multi-core Processors. In *28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 141–151, 2016. doi:10.1109/ECRTS.2016.19.

24 Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. Bus-Aware Multicore WCET Analysis through TDMA Offset Bounds. In *23rd Euromicro Conference on Real-Time Systems*, pages 3–12, 2011. doi:10.1109/ECRTS.2011.9.

25 Hyoseung Kim, Dionisio De Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding memory interference delay in COTS-based multi-core systems. In *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–154, 2014. doi:10.1109/RTAS.2014.6925998.

26 Hyoseung Kim and Ragunathan (Raj) Rajkumar. Predictable Shared Cache Management for Multi-Core Real-Time Virtualization. *ACM Trans. Embed. Comput. Syst.*, 17(1), 2017. doi:10.1145/3092946.

27 Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, 2009.

28 Tomasz Kloda, Marco Solieri, Renato Mancuso, Nicola Capodieci, Paolo Valente, and Marko Bertogna. Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–14, 2019. doi:10.1109/RTAS.2019.00009.

29 Ondrej Kotaba, Jan Nowotsch, Michael Paulitsch, Stefan M Petters, and Henrik Theiling. Multicore in real-time systems–temporal isolation challenges due to shared resources. In *16th Design, Automation & Test in Europe Conference and Exhibition*, 2013.

30 Andreas Löfwenmark and Simin Nadjm-Tehrani. Understanding Shared Memory Bank Access Interference in Multi-Core Avionics. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 55 of *Open Access Series in Informatics (OASIcs)*, pages 12:1–12:11, 2016. doi:10.4230/OASICS.WCET.2016.12.

31 Tamara Lugo, Santiago Lozano, Javier Fernández, and Jesus Carretero. A survey of techniques for reducing interference in real-time applications on multicore platforms. *IEEE Access*, 10:21853–21882, 2022. doi:10.1109/ACCESS.2022.3151891.

32 Claudio Maia, Geoffrey Nelissen, Luis Nogueira, Luis Miguel Pinho, and Daniel Gracia Perez. Schedulability analysis for global fixed-priority scheduling of the 3-phase task model . In *IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10, 2017.

33 Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. Real-time cache management framework for multi-core architectures. In *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 45–54, 2013. doi:10.1109/RTAS.2013.6531078.

34 José Martins, Adriano Tavares, Marco Solieri, Marko Bertogna, and Sandro Pinto. Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems. In *Workshop on Next Generation Real-Time Embedded Systems (NG-RES)*, volume 77, pages 3:1–3:14, 2020. doi:10.4230/OASICS.NG-RES.2020.3.

35 José Martins and Sandro Pinto. Shedding Light on Static Partitioning Hypervisors for Arm-based Mixed-Criticality Systems. In *IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 40–53, 2023. doi:10.1109/RTAS58335.2023.00011.

36 Paolo Modica, Alessandro Biondi, Giorgio Buttazzo, and Anup Patel. Supporting temporal and spatial isolation in a hypervisor for ARM multicore platforms. In *IEEE International Conference on Industrial Technology (ICIT)*, pages 1651–1657, 2018. doi:10.1109/ICIT.2018.8352429.

37 Afonso Oliveira, Gonçalo Moreira, Diogo Costa, Sandro Pinto, and Tiago Gomes. IA&AI: Interference Analysis in Multi-core Embedded AI Systems. In *Data Science and Artificial Intelligence*, pages 181–193, 2025.

38 Claire Pagetti, Julien Forget, Heiko Falk, Dominic Oehlert, and Arno Luppold. Automated generation of time-predictable executables on multicore. In *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, RTNS '18, pages 104–113. Association for Computing Machinery, 2018. doi:10.1145/3273905.3273907.

39 Rob Palin, David Ward, Ibrahim Habli, and Roger Rivett. Iso 26262 safety cases: Compliance and assurance. In *6th IET International Conference on System Safety*, pages 1–6, 2011.

40 Sandro Pinto, Jorge Pereira, Tiago Gomes, Adriano Tavares, and Jorge Cabral. LTZVisor: TrustZone is the Key. In *29th Euromicro Conference on Real-Time Systems (ECRTS)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:22, 2017. doi:10.4230/LIPICS.ECRTS.2017.4.

41 Ralf Ramsauer, Jan Kiszka, Daniel Lohmann, and Wolfgang Mauerer. Look Mum, no VM Exits! (Almost). *CoRR*, 2017.

42 Andreas Schranzhofer, Jian-Jia Chen, and Lothar Thiele. Timing analysis for TDMA arbitration in resource sharing systems. In *16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 215–224, 2010. doi:10.1109/RTAS.2010.24.

43 Gero Schwäricke, Tomasz Kloda, Giovani Gracioli, Marko Bertogna, and Marco Caccamo. Fixed-Priority Memory-Centric Scheduler for COTS-Based Multiprocessors. In *32nd Euromicro Conference on Real-Time Systems (ECRTS)*, volume 165 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:24, 2020. doi:10.4230/LIPICS.ECRTS.2020.1.

44 Ikram Senoussaoui, Houssam-Eddine Zahaf, Giuseppe Lipari, and Kamel Mohamed Benhaoua. Contention-free scheduling of PREM tasks on partitioned multicore platforms. In *IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, 2022. doi:10.1109/ETFA52439.2022.9921531.

45 Alejandro Serrano Cases, Juan M Reina, Jaume Abella Ferrer, Enrico Mezzetti, and Francisco Javier Cazorla Almeida. Leveraging hardware QoS to control contention in the Xilinx Zynq UltraScale+ MPSoC. In *33rd Euromicro Conference on Real-Time Systems (ECRTS)*, volume 196, pages 3–1, 2021.

46 Theo Ungerer, Francisco Cazorla, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Christine Rochange, Eduardo Quiñones, Mike Gerdes, Marco Paolieri, Julian Wolf, Hugues Cassé, Sascha Uhrig, Irakli Guliashvili, Michael Houston, Floria Kluge, Stefan Metzlaff, and Jorg Mische. Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability. *IEEE Micro*, 30(5):66–75, 2010. doi:10.1109/MM.2010.78.

47 Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Heechul Yun, and Marco Caccamo. Global Real-Time Memory-Centric Scheduling for Multicore Systems. *IEEE Transactions on Computers*, 65(9):2739–2751, 2016. doi:10.1109/TC.2015.2500572.

48 Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166, 2014. doi:10.1109/RTAS.2014.6925999.

49 Heechul Yun, Rodolfo Pellizzon, and Prathap Kumar Valsan. Parallelism-aware memory interference delay analysis for COTS multicore systems. In *27th Euromicro Conference on Real-Time Systems*, pages 184–195, 2015. doi:10.1109/ECRTS.2015.24.

50 Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory Access Control in Multiprocessor for Real-Time Systems with Mixed Criticality . In *24th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 299–308, 2012. doi:10.1109/ECRTS.2012.32.

51 Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, 2013. doi:10.1109/RTAS.2013.6531079.

52 Matteo Zini, Giorgiomaria Cicero, Daniel Casini, and Alessandro Biondi. Profiling and controlling I/O-related memory contention in COTS heterogeneous platforms. *Software: Practice and Experience*, 52(5):1095–1113, 2022. doi:10.1002/SPE.3053.